Avantgarde Finance

Appendix III - ISAE 3000 Report Smart Contract Code Assessment

Enzyme Protocol

PwC Switzerland - 19.01.2021

Contents

Avantgarde Finance
12 New Fetter Lane
EC4A 1JP London
United Kingdom

1 Executive Summary

First and foremost we would like to thank Avantgarde Finance and Enzyme Council for giving us the opportunity to assess the current state of their Enzyme Protocol system. We very much appreciated the collaboration with the development team, who were efficient and helpful throughout the process. This document is the Appendix III (Smart Contract Code Assessment) of the ISAE 3000 report, dated January 19th, 2021 and outlines the findings, limitations, and methodology of our assessment.

Tests and Documentation

The project comes with an extensive suite of smart contract tests covering the breadth of functionality intended. There is good documentation available to get an understanding of the system. The functional specification of the core system is available, although it could be extended.

Correctness Findings

We have listed below the three most significant correctness issues uncovered. Correctness issues cover unintended or incorrect behavior of the smart contracts.

- Buyable Share Quantity Incorrectly Calculated
- ManagementFee Initializes lastSettled with 0
- Asset With Unchanged Balance After Call to Adapter handled Incorrectly*

For the complete list of correctness issues, please refer to the Findings starting on page 17.

Design Findings

Listed below are the two most severe design issues uncovered.

- redeemShares() Fails for Fees With SettlementType Direct or Burn
- sharesActionTimelock Can Be Set to Arbitrary Value

For the complete list of design issues, please refer to the Findings starting on page 17.

Security Findings

No major security findings have been uncovered. All security issues are of severity low. For the complete list of security issues, please refer to the Findings starting on page 17.

Conclusion

Overall we have no significant concerns regarding the launch of the Enzyme Protocol. It provides significant improvements in comparison with the previous version. We found that the protocol design and the reviewed code were of consistently high quality. Note that future releases are planned. This report only covers the initial release and provides no security statements regarding future releases.

We hope that this assessment provides you with an insight into the state of the current implementation. We are happy to receive questions and feedback to improve our service.

Yours sincerely,

PricewaterhouseCoopers AG

Report without signature

Andreas Eschbach	Hubert Ritzdorf

Zurich, January 19th, 2021

2 Assessment Overview

In this section we briefly describe the overall structure and scope of the engagement including the code commits which are referenced throughout this report.

2.1 Scope

The general scope of the assessment is set out in our engagement letter with Avantgarde Finance dated November 9, 2020. The assessment was performed on the source code files inside the Enzyme Protocol repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	17 November 2020	a26df25e2fd47152fca768fffb2676f07fcaeb1c	Initial Version
2	2 December 2020	b2687c610b738f4b0b823c0dbee220322b812d32	Version 2
3	23 December 2020	497d76c027317f2ffd1d2b8df5a9b527e2f04ccf	Version 3
4	8 January 2021	d0ad417426a4bf86c66560fc07a759085c0c7d51	Version 4
5	11 January 2021	c91d25afe23286704f446d6e52ac2f1d92243607	Version 5

For the solidity smart contracts, the compiler version `0.6.8` was chosen. After the intermediate report the compiler version was updated to 0.6.12.

2.1.1 Excluded from scope

Vague statements and statements about the behavior of subsequent releases in the documentation are excluded. Notably this includes the statements that future release may allow arbitrary extensions which interact with code present in this release. For the purposes of this review all extensions are trusted. This report gives no statement about the interaction of the current code with untrusted extensions.

The security of the individual services which the adapters connect to (Uniswap etc.) is out of scope. Loss of funds within these services could lead to loss of funds of a fund.

While the review of **Version 1** was ongoing, the developer team independently found some bugs that were fixed for subsequent versions. As these were first discovered by the development team, they are not listed in this report.

The contracts in `mocks` are not part of the scope of this review.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview. At the end of this report section we have added subsections for each of the changes accordingly to the versions. Furthermore, in the findings section we have added a version icon to each of the findings to increase the readability of the report.

Enzyme Protocol (formerly known as Melon Protocol v2) allows decentralized on-chain asset management. The protocol aims to provide a customizable and safe environment for users to invest in funds managed by third parties. Anyone may set up a fund where users can invest.

2.2.1 Core

The implementation logic of the funds is upgradable. To achieve upgradability, most contracts are part of a `release`. These contracts can change in subsequent releases. The `persistent` contracts are non-upgradable. One persistent `Dispatcher` contract maintains the global state of the system and facilitates migrations of the funds to new releases. Further persistent contracts are the `VaultProxy`, where the assets of a fund are held, and the basic implementation of the vault logic / storage, which needs to be inherited by the new `VaultLib` release.

Each fund owner is free to decide whether or not to upgrade.

2.2.2 Infrastructure Contracts

These contracts provide functionality used e.g., by extensions. It includes `PriceFeeds` and a `ValueInterpreter`. For primitives the `ChainlinkPriceFeed` is the single source of truth.

For derivates following price feeds are available:

- `ChaiPriceFeed`
- `CompoundPriceFeed`
- `UniswapV2PoolPriceFeed`
- `AggregatedDerivativePriceFeed` - Main derivative price feed contract. All individual derivative price feeds listed above are registered in this contracts.

The aggregator dispatches the query to the corresponding pricefeed of the respective asset.

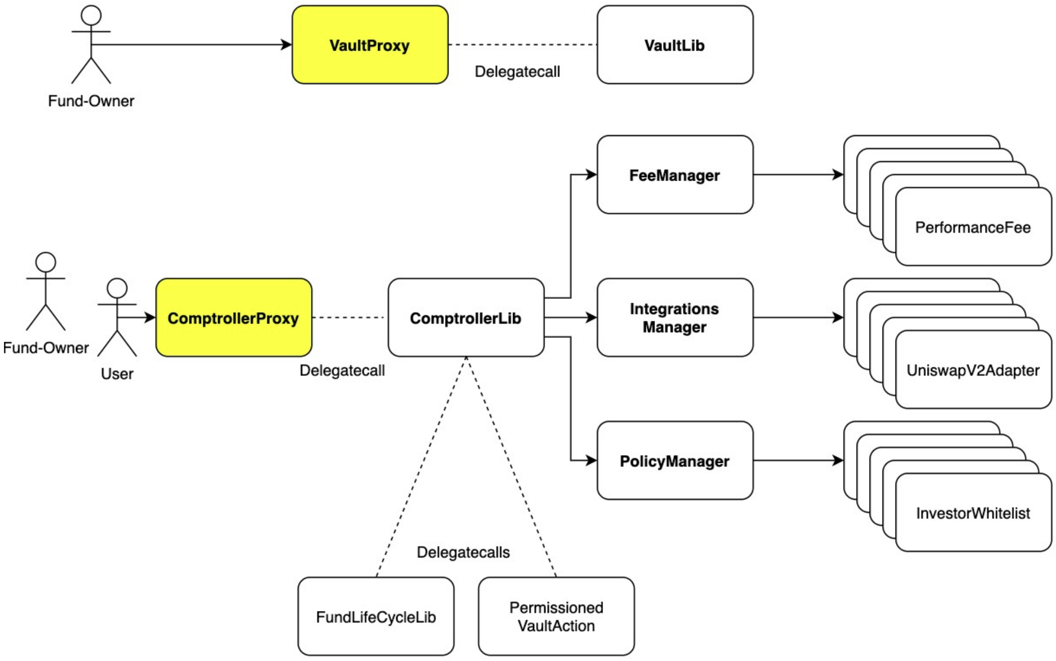
The primitive and aggregated derivative price feed play an important role regarding the security of the system: Both expose `isSupportedAsset()`, only supported assets can be used for actions (e.g. trading) through the `IntegrationManager`. Only primitives can be the denomination asset of a fund.

2.2.3 Fund Deployer

The `FundDeployer` handles deployment of new funds and migration of funds to new releases. The `FundDeployer` contract of the active (most recent) release is registered in the `Dispatcher` as `currentFundDeployer`. Only this `FundDeployer` can be used to deploy or migrate funds.

2.2.4 Funds

A fund consists of two per fund proxy contracts: a `VaultProxy`, where the funds are held, and a `ComptrollerProxy`, which facilitates interactions for the users and the fund owner by maintaining the addresses of the extensions that implement this logic.



All implementation contracts are deployed once per release and shared by all funds. While the `VaultProxy` stays persistent for a fund, the `ComptrollerProxy` of a fund changes after an upgrade.

The `Accessor` role in the `VaultProxy` is the `ComptrollerProxy` of the fund.

The `FeeManager`, the `IntegrationManager` and the `PolicyManager` are so-called extensions. These Extensions implement specific parts of the logic of the `Comptroller` which makes calls on them. Extensions themselves have additional fees/adapters/policies defined in separate contracts which can be activated individually per fund.

Users, also called `investors`, can invest in funds by buying shares paid in the denomination asset of the fund. In this release shares of a fund are non-transferable. Shares can be redeemed at any time given the minimum timelock between action on shares (buying / redeeming) by the user has passed. The protocol strives to guarantee 24h/7 redeemability with no party being able to block redemption. Redemption of shares pays out the corresponding amount of each asset held by the fund.

2.2.5 Extensions

Extensions extend the logic of the comptroller at specific hooks or when called via the comptroller. Three extensions exist, the `FeeManager`, the `PolicyManager` and the `IntegrationManager`. Each of the extensions makes use of so-called plugins. These plugins are the individual fees managed by the `FeeManager`, the individual policies managed by the `PolicyManager` and the adapters to 3rd party systems handled by the `IntegrationManager`.

Different funds share the same extension contracts. Extension manage the configuration for individual funds in their storage.

An example of interaction with the extension is the following: The `Comptroller` makes a request to the `FeeManager` which calls the `Comptroller` back with some results. Then, the `Comptroller` through the `PermissionVaultLib` checks whether it can execute the action on the `Vault` and executes it.

2.2.5.1 Integration manager

The integration manager allows exchanging a fund's assets via so-called `Adapters` to DeFi protocols.

An `Adapter` may implement one or more functions such as:

- `takeOrder(address,bytes,bytes)`
- `lend(address,bytes,bytes)`
- `redeem(address,bytes,bytes)`
- `addTrackedAssets(address,bytes,bytes)`

Most importantly an `Adapter` must feature a `parseAssetsForMethod(bytes4 _selector, bytes calldata _encodedCallArgs)` function decoding the call arguments to the expected assets to receive from a call on the integration adapter. This function must also return whether the tokens to be used for the action should be transferred to the adapter or if an allowance to the adapter should be given.

A call to an adapter of the `IntegrationManager` starts with a call to the `callOnExtension` function of the `Comptroller`. This function features reentrancy protection and the modifier `allowsPermissionedVaultAction`: This sets a boolean to `true` allowing state changing calls to the vault proxy until this function terminates, where the boolean is reset to `false`. The call is forwarded to the `IntegratinManager` where access control is enforced. Only the owner or an authorized user for that comptroller's instance can execute such an action.

Next the policies registered for `PreCallOnIntegration` are validated before the call on the adapter is prepared. The `IntegrationManager` has access to following state-changing vault actions:

- `ApproveAssetSpender`
- `AddTrackedAsset`
- `RemoveTrackedAsset`
- `WithdrawAssetTo`

A check ensures that the used `Adapter` is registered before the calldata are decoded using the adapters's `parseAssetsForMethod` function. The returned values are used after some basic sanity checks, most importantly a check ensures that only supported incoming assets are processed. At this point, the expected outgoing and minimum incoming amounts are known. The assets to be spent are now granted to the adapter.

Next the call to the integration adapter is executed. The only constraint on this call is that after this call a maximum of the amounts specified for the outgoing assets has been spent and that at minimum the listed amounts for the incoming assets have been received. This constraint on the balances of the `Vault` is enforced by the `IntegrationManager` after the call returns. No balances should remain with the `Adapter` after a transaction, all adapters are shared between all funds.

Finally the policies registered for `PostCallOnIntegration` are validated.

In the current release following integration adapters are available:

- `ChaiAdapter`: Allows lending `Dai` for `Chai` and redeeming using `lend()` and `redeem()`,
- `CompoundAdapter`: Allows lending and redeeming tokens to `Compound` using `lend()` and `redeem()`
- `KyberAdapter`: Allows trading on Kyber using `takeOrder()`

- `UniswapV2Adapter`: Allows lending, redeeming and trading assets on Uniswap using `lend()`, `redeem()` and `takeOrder()`
- `ZeroExV2Adapter`: Allows trading on `ZeroEx` using `takeOrder()`
- `TrackedAssetsAdapter`: This is a special adapter allowing direct access to add assets to the list of tracked assets.

2.2.5.2 Policy Manager

Certain hooks allow state validation via the so-called policies while executing actions such as buying shares or making an exchange in the `IntegrationManager`. No policies can be applied while redeeming shares. Policies registered for a fund are invoked at their registered hook and are expected to revert the transaction if their rule is broken.

The `PolicyManager` has no access to state-changing vault actions.

Following hooks are available in the current release:

- `PreBuyShares`:

Executed before shares are bought. Following parameters are passed to the policies: `_buyer`, `_investmentAmount`, `_minSharesQuantity`, `_gav`

- `PostBuyShares`:

Executed after shares are bought. Following parameters are passed to the policies: `_buyer`, `_investmentAmount`, `_minSharesQuantity`, `_gav`

- `PreCallOnIntegration`:

Executed before a call to an adapter of the `IntegrationManager`. Following parameters are passed to the policies: `_adapter`, `_selector`

- `PostCallOnIntegration`:

Executed after a call to an adapter of the `IntegrationManager`. Following parameters are passed to the policies: `_adapter`, `_selector`, `_incomingAssets`, `_incomingAssetAmounts`, `_outgoingAssets`, `_outgoingAssetAmounts`

The hooks are divided in hooks related to `buy-shares` and hooks related to `call-on-integration`. Some policy may have settings which have to be initialized.

Policies related to `call-on-integration` can only be set upon creation/migration of a fund while `buy-shares` policies additionally may be enabled/disabled or have their setting's updated by a fund's owner at any time.

Following policy implementations are available in the current release:

`call-on-integration`:

- `AssetWhiltelist/Blacklist`: represents a set of white/blacklisted assets
- `AdapterWhitelist/Blacklist`: represents a set of white/blacklisted adapters
- `Maxconcentration`: the presence of each asset in the fund should not exceed a proportion of the total

value of the fund.

`buy-shares`:

- `InvestorWhitelist`: Only whitelisted investor may invest

2.2.5.3 Fee Manager

The Fee Manager implements the logic related to fees. It accepts hooks from the comptroller. It then calls particular fee plugins. All fee plugins implement a settle function and may feature an optional update function. As their names suggest, `update` updates the state of a fee and `settle` applies the fee to the assets.

Fees:

- EntranceRateBurnFee: These are entrance fees that are burned from the user's balance.
- EntranceRateDirectFee: These are entrance fees that are transferred to the owner of the fund.
- ManagementFee: This fee is applied as a proportion to the quantity of assets a fund owns. It is proportional to the time that has passed from the previous settlement.
- PerformanceFee: This fee is based on the funds performance, measured in the funds share price. Rewards are due if the share price increases. This fee pays no reward if the share price falls below the previous watermark. The settlement of this fee is handled as "outstanding": shares outstanding are held by the `VaultProxy` and can only be claimed by the owner after a certain crystallization period or upon migration of the fund to a new release.

The `FeeManager` has access to following state-changing vault actions:

- BurnShares
- MintShares
- TransferShares

Hence, fees can only be paid in shares of the fund.

2.2.6 Roles and Trust Model

The roles of the system and their trust assumptions are:

2.2.6.1 Of the overall system:

The owner of the Dispatcher: This is the MTC Council who is fully trusted to act honestly and correctly.

All assets interacting with the Funds are trusted: The price feeds act as whitelisting authority.

In the current release all adapters interacting with the extensions (`IntegrationManager`, `FeeManager`, `Policies`) are audited and trusted.

2.2.6.2 Of a single fund:

`owner` of the fund: Owns the fund. It cannot act arbitrarily upon the VaultProxy but is allowed to execute the registered adapters of the IntegrationsManager and may collects fess if a corresponding rule is active in the FeeManager. The `owner` is an unknown third party. Users investing in the fund trust the `owner` to manage the wealth of the fund. The owners action are limited by the registered adapters of the IntegrationsManager and the active policies guarding interactions of the owner to these adapters. Furthermore the owner has administrator capabilities and is able to override a pause imposed by the owner of the Dispatcher and to migrate the fund to a new release. He is trusted to override the pause only when the bug in question is not affecting the fund he manages. Should a release wide pause be lifted, the fund manager is responsible to reset the override pause boolean in order to avoid overriding the pause immediately should the same release be paused again.

`user / investor` is untrusted. He can invest in funds and redeem his shares at any time. A user is supposed to carefully check the active policies and adapters before investing in a fund as well as the changes taking place during migrations.

`migrator` of the fund: it is allowed to execute a migration.

`accessor` of the fund: this corresponds to the comptroller which is allowed to make changes to the state of the fund. The accessor deploys and handles the extension contracts.

users authorized by the `Comptroller`: defines more users who are allowed to invoke transactions via the `Comptroller`.

`allowedBuySharesCalllers`: as the name suggests these are users who are allowed to buy shares i.e. invoke `buyShares`.

Some extensions define their own roles:

- whitelisted investors: defined by the PolicyManager
- whiltelisted and blacklisted adapters: these are adapters that are allowed or not allowed to be used by the Integration manager.

2.2.6.3 Regarding the Price Feeds

For the price feeds the following trust assumptions exist:

- The primitive price feeds are trusted and the returned values are always assumed to be correct.
- The derivate price feeds are trusted to deliver the right amount of underlyings.
- The price feeds are assumed to take potential illiquidity into account correctly by pricing in exchange rate changes for certain volumes.
- All price feeds, primitive and derivate price feeds, are expected to be flash-loan resistant. The Enzyme Council has to select them accordingly.
- The price feed operator are trusted not to upgrade or reconfigure their price feeds in such a way that they become incompatible.
- The developers are trusted to correctly integrate the price feed outputs, which have different formats, into the protocol.

2.2.6.4 Migration of Funds to a new release

Upon migration to a new release all parameters of the fund can be freely specified by the owner. The trust assumptions is that the parameters of the new fund are duly inspected by the funds investors during the migration timelock. It is planned that the frontend facilitates the comparison of the funds configurations.

2.2.7 Version 2

The main changes / additions of this version are:

- Three new price feeds, the `SynthetixPriceFeed`, the `UniswapV2PoolPriceFeed` and the `WdgldPriceFeed` have been added.
- Two new adapters, the `ParaswapAdapter` and the `SynthetixAdapter`. The `SynthtixAdapter` is notably different from all other adapters. Only Synths can be traded through this adapter and after a trade there is a time lock on transfers of token traded for 180 seconds. This value may change. **Redeeming shares is not possible during this time as the failing transfer will revert the transaction.** After the timelock is over transfers are possible again. For access to the full balance of this asset the exchange must be settled. This can be done by anyone by calling `settle()` on the synthetix exchange contract passing the suitable arguments. For the `SynthetixAdapter` to be able to exchange synths one behalf of a `VaultProxy`, the owner of the `FundDeployer` must add the call `approveExchangeOnBehalf()` of the synthetix exchange to the list of `registerVaultCalls`. Each fund owner can then approve the adapter to exchange the synths.
- As vaults can now hold Synths, certain changes to the system are required. A new infrastructure contract `AssetFinalityResolver` provides the required functionality, the `IntegrationManager` and the `ComptrollerLib` now makes use of this functionality to finalize the balances before handling balances.
- Three new policies: A `MinMaxInvestment` policy enforcing rules on the investment amount when buying shares and `GuaranteedRedemption`, a policy which can inhibit certain adapters to be used during a specific time window during a day. This is required as the new SynthetixAdapter can do trades which do not settle immediately and share redemptions is inhibited before the action finalizes. This daily time window guarantees users can redeem their shares and no tokens are timelocked during this time. The native implementation of `allowedBuySharesCallers` in `ComptrollerLib/FundLifecycleLib` has been removed. A new `BuySharesCallerWhitelist` policy has been added which replaces the removed functionality.
- The calculation of the fee in the implementation of the `ManagementFee` was updated.
- The hooks `BuySharesSetup` and `BuySharesCompleted` have been added to the `Fee` and `Policy` hooks.
- The payout asset amounts due when redeeming shares are now handled differently: Due to the Synths the vault's balance may has to be finalized first. The logic when all remaining shares are redeemed and the handling of removing a tracked asset when all balance is transferred has been changed.
- The `ChainlinkPriceFeed` no longer checks the timestamp of the rate.

2.2.8 Version 3

The main functional changes / additions of this version are:

- Introduction of the (optional) `SharesRequestors` contracts: Investors record their intention to buy shares in the shares requestor contract, these requests are then executed by so called executors.
- The timestamp at which a migration request is executable is now stored rather than the time of the request. Hence, the timelock at the time of the signaling applies.
- `ComptrollerLib.callOnExtension()` no longer allows to call the `PolicyManager`.
- The limit of tracked assets is now hardcoded to `20` and the denomination asset is now always a tracked asset. `A new SpendAssetsHandleType Remove` has been introduced and the `TrackedAssetsAdapter` now supports removing assets with zero balance. Additionally the `IntegrationManager` now allows `auth users` to directly add/remove tracked assets where the vault holds a balance of `0` of the asset.
- Buying shares is no longer possible while a migration is pending.
- The `SharesActionTimelock` is no longer applied when a migration is pending.
- When invoking the contious fee hook manually it is no longer possible to specify the fees to be settled: All fees implementing the `continuous` hook are now settled in their registered order.
- The pricefeeds calculate the underlying value directly instead of returning the rate.

2.2.9 Version 4

The main functional changes / additions of this version are:

- Pricefeeds now convert amounts rather than providing rates
- The formula used in the `PerformanceFee` has been simplified
- The default shares token symbol has been changed from `` MLNF``to ``ENZF ``
- The `FundDeployer` now allows creation/migration of funds only when the `ReleaseStatus` is live

3 Limitations and use of report

3.1 Inherent limitations

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, PwC Switzerland has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts associated with the items defined in the engagement letter on whether it is used in accordance with its specifications by the user meeting the criteria predefined in the business specification. We draw attention to the fact that due to inherent limitations in any software development process and software product an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third party infrastructure which adds further inherent risks as we rely on the correct execution of the included third party technology stack itself. Report readers should also take into account the facts that over the life cycle of any software product changes to the product itself or to its environment, in which it is operated, can have an impact leading to operational behaviours other than initially determined in the business specification.

3.2 Restriction of use and purpose of the report

Our report is prepared solely for Avantgarde Finance for use in connection with the purpose as described in the preceding paragraph. We do not, in giving our opinion, accept or assume responsibility or liability for any other purpose or to any other parties to whom our report is shown or into whose hands it may come.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severities. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved, have been moved to the Resolved Findings section. All of the findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	0

Medium-Severity Findings	4

- ChainLink Integration Code Partially Corrected Acknowledged
- Assets With Composite Balances Risk Accepted
- PostCallOnIntegration Policy Validation After SynthetixAdapter Risk Accepted
- redeemShares() Fails for Fees With SettlementType Direct or Burn Risk Accepted

Low-Severity Findings	6

- Time Window of GuaranteedRedemption Risk Accepted
- Drain Additional Assets Risk Accepted
- No Minimum sharesActionTimelock Enforced Risk Accepted
- Out-of-gas May Prevent Redeeming Shares Risk Accepted
- Theoretical Reentrancy During Migration Risk Accepted
- Tokens With Multiple Entrypoints Risk Accepted

5.1 ChainLink Integration

Design Medium Version 2 Code Partially Corrected Acknowledged

For the primitive pricefeeds, ChainLink price oracles are used. However, the ChainLink API that is currently being used is deprecated:

https://docs.chain.link/docs/deprecated-aggregatorinterface-api-reference

The new API can be found here:

https://docs.chain.link/docs/price-feeds-api-reference

In Version 2 checks for the timestamps of the ChainLink oracles were removed. Hence, stale prices might be used for the calculation. This can result in a loss of funds, however, avoids a potential lock-in.

Acknowledged:

While it is true that there is a more recent API, Avantgarde Finance has confirmed with ChainLink that the `latestAnswer()` and `lastestTimestamp()` functions will always be available on all aggregator proxies. Using `latestAnswer()` saves a substantial amount of gas and is suitable for the intended use.

Code partially corrected:

Stale rates are planned to be handled through off-chain monitoring. Upon notification, the Enzyme Council would remove the affected assets. To further mitigate this issue, a `removeStalePrimitives()` function has been added to the `ChainLinkPriceFeed` contract through which anyone can remove an asset in case its rate is outdated.

5.2 Assets With Composite Balances

Design Medium Version 1 Risk Accepted

For some tokens balanceOf() is not sufficient to query the token balance of the vault. There are cases where the actual balance is a composite of two values. One example are COMP tokens. Vaults will hold COMP tokens directly, which can be queried using balanceOf(), but vaults are also entitled to accumulating COMP tokens that need to be claimed first. Similarly, there could be other token balances, that cannot be entirely calculated with balanceOf(), e.g., due to certain staking opportunities.

For the correct calculation of the gross asset value of the fund these assets need to be handled differently from others. For such assets, the composite balance must be considered.

As this is a non-standard behavior of these tokens, no default solution for all such tokens exists.

The appropriate handling depends on the operational requirements. If it should be possible to add a “complicated” asset without migrating to a new release, the solution could make use of extensions implementing the calculation of the actual balance for such special assets (similarly to derivative price feeds). This, however, significantly increases the gas costs when calculating the GAV due to the additional contract calls. A more lightweight solution could be to implement the calculations directly when calculating the GAV. Such a solution may be possible if no “complicated” assets are added without migrating to a new release.

The correct calculation of the gross asset value is of high importance to protect the investors of the fund: Discrepancies in the actual value of the fund and the calculated gross asset value can be exploited as you mentioned.

Risk accepted:

Avantgarde Finance acknowledges the issue and states that this particular issue only applies to `COMP` in the currently planned asset universe. Potential solutions may exist, however, significantly increase the gas consumption of the transaction and hence are simply not workable as short-term solution. Overall, this is not seen too critical in the specific case of `COMP` and likely any other externally accruing asset as the arbitrage opportunity that it presents is, at worst, comparable with the known opportunity to arbitrage ChainLink prices.

5.3 `PostCallOnIntegration` Policy Validation After `SynthetixAdapter`

Design Medium Version 1 Risk Accepted

After a call on an adapter of the `IntegrationManager`, the policies registered for the `PostCallOnIntegration` are validated. Following arguments are passed:

```
IPolicyManager(POLICY_MANAGER).validatePolicies(
    msg.sender,
    IPolicyManager.PolicyHook.PostCallOnIntegration,
    abi.encode(
        _adapter,
        _selector,
        _incomingAssets,
        _incomingAssetAmounts,
        _outgoingAssets,
        _outgoingAssetAmounts
    )
```

After a call to the `SynthethixAdapter` the situation is special. The `incomingAssetAmount` of the synth is the current balance that has just been queried. However, note that there is a timelock of 180 seconds until the exchange is finalized. Upon settlement the actual amount can change. Hence, the value validated by the policies is not the actual outcome of the action. This is inherent to the synthetics. Users of the system need to understand these implications which should be highlighted in the dApp.

Risk accepted:

Avantgarde Finance states:

> There are a number of quirks regarding Synths that need to be conveyed to end users. The point about how Synths affect policy management is duly noted and will be one of the items presented in our frontend solution.

5.4 `redeemShares()` Fails for Fees With SettlementType `Direct` or `Burn`

Design Medium Version 1 Risk Accepted

To redeem shares users call `redeemShares()` implemented in `ComptrollerLib` through the `ComptrollerProxy`. This invokes:

```
__redeemShares(
           msg.sender,
           ERC20(vaultProxy).balanceOf(msg.sender),
           new address[](0),
           new address[](0)
       );
```

The users balance is used as amount to be redeemed. After the `__preRedeemSharesHook` has been executed, a check ensures that the user has enough shares for the redemption:

```
 __preRedeemSharesHook(_redeemer, _sharesQuantity);
}

...

// Check the shares quantity against the user's balance after settling fees
require(
    _sharesQuantity <= sharesContract.balanceOf(_redeemer),
    "__redeemShares: Insufficient shares"
);
```

However, if a fee settling on `PreRedeemShares` uses `SettlementType.direct` or `SettlementType.burn` where the fee is paid from the user's share balance, this check will revert the transaction as the user will have an insufficient balance after having paid the fees.

Note that in the current release under review only two fees are active on the `PreRedeemShares` hook, the `PerformanceFee` and the `ManagementFee`. Neither of them use `SettlementType` `direct` or `burn` and hence these are not affected by the issue described above.

Risk accepted:

Avantgarde Finance states:

> This is a known limitation of the way fee management currently works, and since only fees sanctioned by the Enzyme Council will be made available for this release, we are planning to simply avoid such fees.

5.5 Time Window of `GuaranteedRedemption`

Design Low Version 2 Risk Accepted

The time window of the `GuaranteedRedemption` policy blocks calls to certain adapters during this time window. The time window repeats every day at the same time. This window can be between 1 second up to 23 hours. Setting the time window to 0 is special and completely disallows call to the adapters registered for this policy.

- Setting the time window to one or a few seconds can only result in no GuaranteedRedemption window as no block is mined containing a timestamp inside the time window
- For the `SynthetixAdapter`, any time window below it's waiting time of 180 seconds results in no GuaranteedRedemption window. A trade done just before the time window starts would span over the time window without transfers being possible during the GuaranteedRedemption window.
- As the time window starts at the same hour every day, a short time window is inconvenient for users across the globe as it might be well past midnight for them.

The minimum time may be reconsidered or documented.

Risk accepted:

This and other similar trust issues will be handled on the frontend by flagging potentially dangerous configurations.

5.6 Drain Additional Assets

Security | Low | Version 1 | Risk Accepted

`ComptrollerLib.redeemSharesDetailed()` allows to specify additional (non-tracked) assets to claim. Intended as protection for investors, this functionality allows any investor to drain the funds balance of the non tracked asset: As non-tracked assets are not included in the calculation of the gross asset value, investors may buy shares without paying for the non-tracked assets. Upon redemption however the investor can get the additional asset on top of his part of the tracked assets. Doing this repeatedly allows to drain non-tracked assets almost for free.

Risk accepted:

Avantgarde Finance states:

> This is an important observation and known limitation, though it does not only apply to specifying additional assets in *redeemSharesDetailed()*: if a fund has an untracked asset that becomes tracked, any investor who enters the fund prior to the tracking of said asset (e.g., mempool frontrunners) will receive a portion of that asset free-of-charge.
>
> This has proven to be a difficulty of airdrops in Melon v1, such as Uniswap's UNI token launch.
>
> It is also a potential attack vector for a malicious fund manager, e.g., buying fund shares immediately before claiming an airdrop and then adding the asset to tracked assets via the *TrackedAssetsAdapter*. Unfortunately, it is impossible keep tabs on the limitless range of untracked tokens that an account receives, and so the free-for-all competition to grab untracked airdrop tokens can only be mitigated through an artful use of policies (which we will consider developing if this becomes a significant issue in practice).

5.7 No Minimum `sharesActionTimelock` Enforced

Security | Low | Version 1 | Risk Accepted

The `sharesActionTimelock` has been introduced to implement a minimum time lock between action on shares, namely buying and redeeming. This prevents buying and redeeming shares within the same transaction, essentially mitigates any kind of attack based on flashloans.

The parameter `sharesActionTimelock` can be set when creating a new fund config. While the documentation correctly describes that this value should be set to a minimum of 1, this is not enforced by the implementation.

Hence, a fund may be created with a timelock on shares actions of 0. In this case the protection against price manipulation using flash loans would be disabled.

Risk accepted:

Avantgarde Finance states:

> Starting with this v2 release, we are striving to create a protocol that is as abstract and unrestrictive as possible at its core, in order to not hamper third party applications that may wish to use the protocol in ways that are different to our primary use case (the flagship frontend).

5.8 Out-of-gas May Prevent Redeeming Shares

Design | Low | Version 1 | Risk Accepted

The Enzyme Protocol aims for 24/7 redeemability and the implementation contains multiple tweaks to ensure shares can always be redeemed even in case of problems e.g. during the calculation of fees due.

The core logic of redemption is implemented in the `__redeemShares` helper of the `ComptrollerLib`. There, provided that the fund is not paused, there is a hook to the fee manager wrapped in a `try/catch` clause. However, such a call to a badly implemented fee manager can consume all the gas. Notice that the out-of-gas exceptions are not caught by the `try/catch` clause. A limited amount of gas may be passed to this call to allow the execution and hence shares redemption to continue even if this call uses all passed gas.

Risk accepted:

Avantgarde Finance states:

> This is a great note for when we start to allow third party fees and need to take further precautions against malicious actions such as DoS attacks (e.g., intentionally consuming all gas during redemption). For this release where all fees are audited and approved by the Enzyme Council, we can forego this protection.

5.9 Theoretical Reentrancy During Migration

Security | Low | Version 1 | Risk Accepted

While a migration is ongoing, which is triggered by the call to `executeMigration` on the new `FundDeployer` several contracts are called. As part of the migration first, the old fee contracts are executed to settle potentially outstanding fees. In the current release these fee contracts are considered trusted, however, in later releases they might not be. Hence, we raise this issue for future releases. These fee contracts could collude with a malicious party and could perform a reentrancy. They could either reenter the `signalMigration` function of the `FundDeployer` or reenter the `cancelMigration` function of the `FundDeployer`. Both cases would lead to an inconsistent state and confusing events.

Later during the migration the new fee contracts, new policy contracts and new integration contracts are called. These contract could similarly perform a reentrancy. They could, however, only reenter the `signalMigration` function. Again this would result in an inconsistent state for the `Dispatcher` where a migration request for an already migrated comptroller would exist.

Risk accepted:

Avantgarde Finance states:

> Nice observation, and noted to revisit when third party plugins (fees, policies, adapter) and extensions are enabled.

5.10 Tokens With Multiple Entrypoints

Security Low Version 1 Risk Accepted

This is more a theoretical issue but has applied to tokens in the past. Nowadays this is a less common issue. Some (very few) tokens have multiple addresses as entry points, e.g. a proxy not using delegatecall and the actual implementation contract.

If a supported token has multiple entrypoint addresses, the `additionalAssets` of `redeemSharesDetailed()` in `ComptrollerLib` can be abused to retrieve more of the very same token.

Hence all listed tokens by the Enzyme Council must have unique entrypoints.

Risk accepted:

Avantgarde Finance states:

> Noted. Since this is exceedingly rare, we can simply add to our checklist for conditions of adding to our asset universe.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	1

- Buyable Share Quantity Incorrectly Calculated Code Corrected

Medium-Severity Findings	4

- Asset With Unchanged Balance After Call to Adapter Handled Incorrectly* Code Corrected
- Unused Optimizer Code Corrected
- ManagementFee Initializes lastSettled With 0 Code Corrected
- sharesActionTimelock Can Be Set to Arbitrary Value Code Corrected

Low-Severity Findings	10

- Misleading Use of the Term 24/7 Redemption Specification Changed
- Description of Flash Attacks in Known Risks Specification Changed
- DoS Possibility When Migrating Funds With Synths and the PerformanceFee Code Corrected
- Exceeding TrackedAssets Limit Code Corrected
- Functions Should Not Be Marked as Payable Code Corrected
- Inefficient Detection Of Double Entries Code Corrected
- Outdated Compiler Version Code Corrected
- Transferable Shares Specification Changed
- Unused Boolean in Modifier Code Corrected
- Creator Can Execute State-Changing Calls to VaultProxy Specification Changed

6.1 Buyable Share Quantity Incorrectly Calculated

Correctness High Version 1 Code Corrected

In case that the denomination asset has a different number of decimals than 18, then the function `__calcBuyableSharesQuantity` which should compute the number of shares an investor receives given a certain `_investmentAmount`, calculates the wrong result. Below we provide an example.

```
uint256 denominationAssetUnit = 10**uint256(_denominationAssetContract.decimals());
return
```

```
    _investmentAmount.mul(denominationAssetUnit).div(
        __calcGrossShareValue(_gav, _sharesContract.totalSupply(), denominationAssetUnit)
    );
```

In our example we assume that the denomination asset has 16 decimals. If the investment amount equals the Gross Asset Value (GAV), then the amount of buyable shares should equal the current total supply of shares as the investor would double the GAV and should hence hold half of the new share supply. However, in the current case the Gross Share Value will be calculated as:

```
return _gav.mul(SHARES_UNIT).div(_sharesSupply);
```

where `SHARES_UNIT = 10**18`. Hence, in our example the formula for the buyable shares will be:

$$BuyableShares = \frac{investmentAmount \cdot totalSupply \cdot 10^{16}}{GAV \cdot 10^{18}} = \frac{totalSupply}{100}$$

Code corrected:

The incorrect formula has been fixed and now uses the `SHARES_UNIT` instead of the `_denominationAssetUnit`. Additionally the test suite has been enhanced and now contains a test case featuring a fund with a non-18 decimal denomination asset.

6.2 Asset With Unchanged Balance After Call to Adapter Handled Incorrectly*

Correctness Medium Version 1 Code Corrected

While the review was ongoing Avantgarde Finance found this issue independently in parallel.

The function `IntegrationManager.__reconcileCoISpendAsets()` processes the spent assets after a call to an adapter was executed. It processes the actual spent amounts and documents whether the balance of a spent asset has increased.

A comment in the code states:

```
If the pre- and post- balances are equal, then the asset is neither incoming nor outgoing.
```

This case is not handled properly by the implementation:

In the first `for` loop, the value of `postCallSpendAsetsBalances[i]` is compared with `_preCallSpendAssetBalances[i]` before the actual post call balance has been queried and stored in this variable. Hence, the post call balance used for this comparison is always 0 and the asset is not skipped in case of equality. This results in an increase of the `increasedSpendAssetsCount` counter:

```
uint256[] memory postCallSpendAssetBalances = new uint256[](_spendAssets.length);
uint256 outgoingAssetsCount;
uint256 increasedSpendAssetsCount;
for (uint256 i = 0; i < _spendAssets.length; i++) {
    // If spend asset's initial balance is 0, then it is an incoming asset.
    // If the pre- and post- balances are equal, then the asset is neither incoming nor outgoing.
    if (
        _preCallSpendAssetBalances[i] == 0 ||
        postCallSpendAssetBalances[i] == _preCallSpendAssetBalances[i]
    ) {
        continue;
    }
```

Later this counter value is used in:

```
increasedSpendAssets_ = new address[](increasedSpendAssetsCount)
increasedSpendAssetAmounts_ = new uint256[](increasedSpendAssetsCount)
```

In the next for loop assets with unchanged balances are properly skipped as `postCallSpendAsetsBalances[i]` has been populated with the actual balance in the meantime. However, then the allocated arrays might be too big and the last entries may remain empty.

Code corrected:

The issue has been fixed with adding separate checks which catch both cases of increase and decrease in balance and ignoring cases where the balance remains the same. For example:

```
if (postCallSpendAssetBalances[i] < _preCallSpendAssetBalances[i]) {
    outgoingAssetsCount++;
} else if (postCallSpendAssetBalances[i] > _preCallSpendAssetBalances[i]) {
    increasedSpendAssetsCount++;
}
```

6.3 Unused Optimizer

Design Medium Version 1 Code Corrected

The project's settings defined in `hardhat.config.js` do not enable the optimizer of the Solidity compiler. Enabling the optimizer will reduce the gas usage and hence transaction fees.

A contract's bytecode can either be optimized for deployment or for execution. The decision on what to optimize for depends on how many times the contract's code is estimated to be executed.

Code corrected:

The solidity optimizer has been enabled. Note that the yul optimizer had to be disabled due to `Yul stack-too-deep` errors.

6.4 ManagementFee Initializes lastSettled With 0

Correctness Medium Version 1 Code Corrected

The lastSettled variable of the ManagementFee is set to 0 upon initialization of the fund settings.

```
function addFundSettings(address _comptrollerProxy, bytes calldata _settingsData)
    ....
{

    ....

    comptrollerProxyToFeeInfo[_comptrollerProxy] = FeeInfo({rate: feeRate, lastSettled: 0});

    ....

}
```

As a result, the fees for the first settlement after migrating to a new release are really high since they are calculated using the previous time the fees were settled.

```
function __calcSettlementSharesDue(

    ....

) private view returns (uint256) {

    ....

    return
        __calcSharesDueWithInflation(
            yearlySharesDueRate.mul(block.timestamp.sub(_prevSettled)).div(RATE_PERIOD),
            _sharesQuantity
        );
}
```

Code corrected:

Upon activation of the fee for a fund, function activateForFund now sets lastSettled to the current timestamp.

6.5 sharesActionTimelock Can Be Set to Arbitrary Value

Design Medium Version 1 Code Corrected

The fund owner can set the sharesActionTimelock to an arbitrary value upon initialization of a new or migrated fund config. This value can exceed the migrationTimelock. This means that a user who has recently made an action such as buying shares may not be able to redeem their assets during the migration timelock if they want to opt out of a migrating fund.

Consider the following scenario: Assume that the migrationTimelock is set to `MTL` and the actionShareTimelock is set to AST. Assume that `MTL` < AST. The user buys shares at point t and at the same point a migration is signalled. Then the user cannot redeem their shares since they have to wait for AST seconds while the migration is executed, earlier, after `MTL` seconds.

Morever, a high value of `sharesActionTimeLock` contradicts with the requirement for 24/7 redeemability, potentially blocking shares redemption indefinitely.

An upper bound for the value of AST could be introduced. Such an upper bound could be set to be half the value of the `migrationTimelock`.

Code corrected:

In the updated implementation buying shares is no longer possible while a migration is pending. This allows to mitigate the issue described above by simply ignoring the `sharesActionTimelock` during redemption when a fund has a pending migration.

6.6 Misleading Use of the Term 24/7 Redemption

Correctness **Low** **Version 2** **Specification Changed**

The usage of the term `24/7 Redeemability` is misleading after the introduction of Synths. The text of the chapter `24/7 Redeemability` in the documentation contains the following sentence:

```
all assets in our assets universe must be freely transferable, or must be accompanied by
a fund policy that explicitly allows or non-continuous redemption availability (e.g. Synths)
```

While this hints that there might be some restriction when Synths are involved, overall the text leaves a strong impression that redemption is guaranteed 24h/7. This does not hold. The only guarantee when the `GuaranteedRedemption` policy is active is that there is a guaranteed time window every 24h when share redemption succeeds as no assets are blocked. Note that due to another issue with this policy (time window can be set too small) this currently doesn't hold even when the policy is active. For more information please refer to issue `Time Window of GuaranteedRedemption`.

Specification changed:

This part of the documentation has been replaced with a new "Continuous Shares Redeemability" page.

6.7 Description of Flash Attacks in Known Risks

Correctness **Low** **Version 1** **Specification Changed**

The description of the `Flash attacks` in chapter `Known risks & mitigations` states:

```
... move the price back and return the loan in one block.
```

Note that attacks using flash loans must be completed within one transaction. If the loan is not repaid at the end of the transaction the transaction reverts.

Specification changed:

The typo was adjusted from "block" to "transaction".

6.8 DoS Possibility When Migrating Funds With Synths and the `PerformanceFee`

Security Low Version 1 Code Corrected

Upon creation or migration of a fund the fees are activated. The activation of the `PerformanceFee` stores the current `GAV` of the fund. The call to `ComptrollerLib.calcGrossShareValue()` has the parameter `_requireFinality` set to true which is required to ensure the correct value of the `GAV` is calculated. However, the calculation will fail if there are synths that cannot be settled yet.

A (theoretical) attacker can repeatedly front-run the migration transaction visible in the transaction pool and exchange a minimum amount of a tracked synths passing the `VaultProxy` of the fund as destination address. This would prevent migration.

It may also happen unintentionally if a trade is done through `SynthetixAdapter` shortly before the migration.

Code corrected:

The share price calculation no longer requires finality.

6.9 Exceeding TrackedAssets Limit

Correctness Low Version 1 Code Corrected

The number of tracked assets of a fund is limited. Adding a tracked asset does not enforce the limit directly, this has to be checked separately. The list of tracked asset may increase after a call on an integration adapter due to newly incoming assets or when shares are bought and paid for in the denomination asset of the fund, which could have been a non tracked asset.

It is known and documented that the `trackedAsssetLimit` can be exceeded if the denomination asset is not tracked and `buyShares()` is executed.

The `IntegrationManager` does following check:

```
require(
    postTxTrackedAssetsCount <= trackedAssetsLimit ||
        postTxTrackedAssetsCount <= _preTxTrackedAssetsCount,
    "__validateTrackedAssetsLimit: Limit exceeded"
);
```

This ensures that the call on the integration adapter did not result in exceeding the `trackedAssetsLimit` but allows the transaction to successfully complete if the `trackedAssetsLimit` was already exceeded before but did not increase during this call.

This may be abused by any authorised caller to the `IntegrationManager` by a fund to add an arbitrary amount of tracked assets:

1. Exchange all funds of the denomination asset into the asset to be added to the list of tracked assets.
2. Buy shares using `buyShares()`. This adds the denomination asset to the list of tracked asset and exceeds the `TrackedAssetLimit`. This behavior is documented and accepted.
3. Exchange all funds of the denomination asset into the asset to be added to the list of tracked assets.

This can be repeated to add an arbitrary amount of assets to the list of tracked assets.

Code corrected:

Three interrelated changes have been made to address this issue.

1. The tracked assets limit is now a constant in the `VaultLib` and is strictly enforced.
2. The denomination asset is now always a tracked asset and is never allowed to be removed from a fund.
3. There are now actions on the `IntegrationManager` for fund owners and auth users to freely add and remove tracked assets that have a zero-balance.

6.10 Functions Should Not Be Marked as Payable

Design | Low | Version 1 | Code Corrected

The functions `createFund` and `buyShares` of the `FundDeployer` and the `Comptroller` respectively are marked as `payable` however they do not handle ether i.e., `msg.value` at any point.

Code corrected:

The `payable` modifier has been removed.

6.11 Inefficient Detection Of Double Entries

Design | Low | Version 1 | Code Corrected

`FundLifecycleLib.init()` adds the addresses of the parameter `_allowedBuySahresCallers` to `allowedBuySharesCallers`. A check prevents duplicate entries:

```
for (uint256 i; i < _allowedBuySharesCallers.length; i++) {
    require(
        i == 0 || !allowedBuySharesCallers.contains(_allowedBuySharesCallers[i]),
        ": buy shares caller already allowed"
    );

allowedBuySharesCallers.add(_allowedBuySharesCallers[i]);
emit AllowedBuySharesCallerAdded(_allowedBuySharesCallers[i]);
}
```

`allowedBuySharesCallers` is a `EnumerableSet.AddressSet`. The `add` function returns a boolean and adds the entry only if it's not already present. This return value could be inspected instead of the separate check using `.contains()`-

---

Code corrected:

As the code related to `allowedBuySharesCaller` has been removed in favor of a policy, this issue no longer exists.

6.12 Outdated Compiler Version

Design Low Version 1 Code Corrected

The project uses an outdated version of the Solidity compiler.

```
pragma solidity 0.6.8;
```

Known bugs in version 0.6.8 are: https://github.com/ethereum/solidity/blob/develop/docs/bugs_by_version.json#L1326

More information about these bugs can be found here: https://docs.soliditylang.org/en/v0.6.8/bugs.html

At the time of writing the most recent Solidity release is version `0.7.5`. For version `0.6.x` the most recent release is `0.6.12` which contains some bugfixes but no breaking changes.

Code corrected:

The compiler version has been updated to 0.6.12.

6.13 Transferable Shares

Security Low Version 1 Specification Changed

In chapter `End Users - Investor` on page 10 the documentation clearly states:

```
Shares are currently non-transferrable.
```

For investors this is true. They cannot transfer their shares as the current version of the VaultLib overrides the transfer/transferFrom and approve function and calls would revert.

Technically however, may be transferred during the settlement of fees. These transfers correctly emit the Transfer event. Hence the statement in the documentation is imprecise.

Transfers of shares would be problematic as they interfere with the sharesActionTimelock functionality. This timelock is required to prevent attacks using flash loans as a minimum time lock of one second would guarantee that buying and redeeming shares cannot be done within one transaction. The current implementation of the sharesActionTimelock` however works per account and could be circumvented by a transfer.

While this issue may not apply in the current release where the available fees only transfer to a restricted set of trusted accounts, this must be carefully considered for future release where untrusted fees or transferable shares may exist.

Specification changed:

The specification was updated to be clear that shares are not freely transferable between investors. An internal note about the effect of transferrable shares would have on the *sharesActionTimelock* has been made.

6.14 Unused Boolean in Modifier

Design Low Version 1 Code Corrected

In the modifier onlyPermissionedAction in PermissionedVaultActionLib, a boolean isValidAction is defined but never unused.

Code corrected:

The unused boolean has been removed.

6.15 Creator Can Execute State-Changing Calls to VaultProxy

Correctness Low Version 1 Specification Changed

In the section about the ComptrollerProxy in chapter ReleaseArchitecture the specification states:

```
All state-changing calls to the VaultProxy must thus pass through the
ComptrollerProxy, making it a critically important bottleneck of access control.
```

VaultLib inherits from VaultLibBaseCore. This contract defines functions setAccessor() and setVaultLib() both which are only accessible for the creator. The creator of the VaultLib is the Dispatcher. These functions are used during migration of the fund.

Hence the comment that all state-changing calls to the `VaultProxy` must pass through the `ComptrollerProxy` is incorrect.

Specification changed:

The imprecise specification has updated to be accurate.

7 Notes

We leverage this section to highlight potential pitfalls which are fairly common when working Distributed Ledger Technologies. As such technologies are still rather novel not all developers might yet be aware of these pitfalls. Hence, the mentioned topics serve to clarify or support the report, but do not require a modification inside the project. Instead, they should raise awareness in order to improve the overall understanding for users and developers.

7.1 Deactivation Of Plugins

Note Version 1

The `FundDeployerOwner`, a very privileged and trusted role, can register and deregister plugins for the 3x Extensions (IntegrationManager, FeeManager and PolicyManager). The documentation contains only one sentence about this:

```
Allowed plugins are all defined on registries in their respective Extensions
```

While this sentence is true, the documentation around this should be expanded and explain / specify how this is actually handled in the implementation.

Removing plugins of the different Extensions have different effects:

- Removing a plugin (adapter) of the `IntegrationManager` has an immediate effect: this adapter cannot be used anymore.
- Removing a plugin (policy) of the `PolicyManager` has a different effect: This policy cannot be registered anymore for new funds but still applies for funds already having this policy enabled.

Deregistering adapters of the `IntegrationManager` may be required for security reasons, e.g. to disable them in case of an issue. However this can be problematic: Adapters lending out funds might be required to retrieve these funds. E.g., a fund lends an amount of tokens through the `lend()` function of the adapter. Later these tokens are reclaimed using the `redeem()` function of the adapter. If the adapter has been disabled, reclaiming the funds through the adapter is no longer possible for the managers or the fund owner.

No tokens are lost if lending assets results in incoming assets of so-called lending tokens. These may be exchanged through another adapter.

7.2 Fee Config and Migration

Note Version 1

The migration process of the fee configuration is not documented. Upon migration of a fund, first a new migrated fund config is created in the `FundDeployer`. This already includes the fee configuration and executes `FeeManager.setConfigForFund()` which call `addFundSettings` on each fee, passing the configuration.

However, the current fund continues to operate and the migration only happens after a minimum timelock has expired. After this minimum timelock, more time may elapse before the actual migration takes place. During this time more fees might be settled.

The specification should clarify that all the fees must evaluate the shares due correctly after migration.

7.3 Fees Before Shares Are Minted

Note Version 1

This issue has no immediate impact in the release under review as no problematic fees exist. However, this problem may arises in fees of further releases.

Fees implement `implementedHooksForSettle_` and `implementedHooksForUpdate_`. The available hooks include `BuySharesSetup` and `PreBuyShares`. These hooks are invoked in `buyShares()` before the shares bought are minted. `ImplementedHooksForSettle_` on these hooks may be problematic. In most cases, settling fees will fail before the shares have been minted.

Furthermore, `SettlementType`, `Mint` and `MintSharesOutstanding` have a special restriction and only work when the `totalSharesSupply` of the fund is non-zero. Settling fees with this `SettlementType` is not possible before the first shares have been minted.

In both cases the funds would be inoperable as investing is not possible.

The dApp should only allow configurating funds with working fee configurations.

7.4 Front-Running Calls to Adapters of the `IntegrationManager`

Note Version 1

The fund owner or authorized users acting as fund managers can trade with the funds asset through Adapters of the `IntegrationManager`. These calls may fail if the funds asset balance reduced unexpectedly as shares are redeemed. Notably this might happen when all balance of a specific asset is to be used through an adapter. Any share redemption before the execution of the transaction through the integration manager will inhibit the trade. Malicious fund investors can do it repeatedly on purpose and effectively perform a denial of service attack on the fund.

7.5 Rich Actors May Undermine Flashloan Protection

Note Version 1

The risk of flash loans and their power for price manipulation is well known and described in the documentation. Flash-loan based attacks are effectively prevented with a time lock between buying and redeeming shares.

However, as the timelock can be as low as one second according to the documentation, rich and influential actors cannot be ruled out as attackers. If such actors can place one transaction at the end of a block and another transaction at the beginning of the next block, they can basically perform all attacks commonly considered as flash-loan attacks. Under-collateralized loans and specific miner services for transaction submission further increase this risk.

7.6 Slippage Protection When Fund Owner Invests

Note **Version 1**

When investing into a fund using `buyShares()` a parameter `_minSharesQuantity` has to be specified. This acts as slippage protection i.e., should the expected amount of shares not be received, the transaction reverts.

```
sharesReceived_ = sharesContract.balanceOf(_buyer).sub(prevBuyerShares);
require(
    sharesReceived_ >= _minSharesQuantity,
    "buyShares: Shares received < _minSharesQuantity"
);
```

`prevBuyerShares` is the balance before the shares are minted but after the `preBuySharesHook` has been executed. However, next to the minting, the `postBuySharesHook` is also invoked which has an effect on the balances before `sharesReceived` is calculated.

A comment in the code hints at this:

```
// The number of actual shares received may differ from shares bought due to
// how the PostBuyShares hooks are invoked by Extensions (i.e., fees)
```

Noteworthy is the corner case when shares are `minted/transferred/burned` to the beneficiary and the beneficiary invests himself. In this case the slippage protection would be incorrect, either `false-positive` or `false-negative`.

No fee available in the current release does this at this hook, the specification however should cover this scenario to prevent potential problems with future fees.

7.7 Third-Party Extensions / Plugins

Note **Version 1**

The documentation hints that future releases may allow arbitrary extensions and plugins for extensions by third parties. In preparation for this, some parts of the code are already implemented, e.g. `PermissionedVaultActionLib`.

This seems tricky especially as the code of untrusted extensions may change if it's code contains a reachable `SELFDESTRUCT`. Destructed contracts that have been deployed using `CREATE2` may be redeployed later with different code.

Fee-Plugins can transfer/mint shares, hence eventually drain all assets of a fund.

Both current implementations of the `FeeManager` and the `PolicyManager` rely on all enabled fees for a fund to be working. Should one revert the whole transaction fails. Currently policies active for a fund cannot be removed.

While this may work for trusted & audited plugins, this may has to be reconsidered if untrusted plugins are allowed.

7.8 Tokens With Transfer Fees

Note Version 1

Some tokens may have a fee on transfers.

It should be ensured that no primitives with transfer fees are added by the MTC. If the denomination asset of a fund has a transfer fee, the calculation of the shares to be issued will not be as expected: The calculation is based on the investment amount. However when the token is transferred to the vault, the investment amount minus the transfer fee is added to the vault's balance of the asset.

For derivates which can be traded via Adapters of the `IntegrationManager` transfer fees may be supported.

7.9 While Buying Shares for Multiple Buyers `sharePrice` Remains Constant

Note Version 2

`ComptrollerLib.buyShares` allows a user to buy shares for many buyers. The GAV and the share price is calculated upfront. While issuing new shares for each buyer, the value of GAV is updated however the share price is not.

```
for (uint256 i; i < _buyers.length; i++) {
    sharesReceivedAmounts_[i] = __buyShares(
        ...
        sharePrice,
        gav,
        ...
    );

    gav = gav.add(_investmentAmounts[i]);
}
```

This may be desirable as the ordering has no impact on the shares received by the individual buyer. However note that the share price may change slightly if recomputed between each buyer: In case of a fee minting or burning shares the total share supply is changed without a corresponding change in the gross asset value of the fund, hence the recalculated share price would be different.

7.10 `Immutable` Keyword

Note Version 1

The `Immutable` keyword is used to set "constant variables" inside the constructor. One example is the `ComptrollerLib`. This code is executed as `DELEGATECALL`. For the code to work as expected, these constants need to be set in the bytecode, not in the contract's storage. This is important as a `DELEGATECALL` executes the code of the callee in it's own (storage-) context.

Currently Solidity does this as expected. The blogpost by the Solidity developer explains how the new `immutable` keyword works:

> While this mechanism may change in the future, currently, during contract creation, a dedicated memory area is reserved for the values of immutables. The constructor code can then store the intended values for the immutables in that memory area. The compiler-generated part of the creation code that is executed after the constructor and returns the contract's runtime code will read back the values of the immutables and insert them into all occurrences in the runtime bytecode.

https://blog.soliditylang.org/2020/05/13/immutable-keyword/

Note that this mechanism may change in the future and this may have implications on code executed as `DELEGATECALL`. When the compiler version used for the project is updated in the future it should be checked if this still works as intended.

7.11 `MaxConcentrationPolicy` May Not Hold With Synthetix

Note **Version 1**

The `MaxConcentrationPolicy` checks the max concentration of the funds assets after a call to the `IntegrationManager`. However with synths, the balance checked at this point is not the final balance which may change upon settlement. Consequently a fund's asset distribution may not adhere to the rules set by the `MaxConcentrationPolicy` after the exchange done through the `IntegrationManager` settles.

When Synths have been introduced, the implementation of the `MaxConcentrationPolicy` has been updated:

```
// Does not require asset finality, otherwise will fail when incoming asset is a Synth
  (uint256 totalGav, bool gavIsValid) = comptrollerProxyContract.calcGav(false);
```

The documentation should clearly highlight the limitations of this policy on interactions with synths. The situation may arise that a funds asset concentration significantly differs from the limits set in the `MaxConcentrationPolicy`.

7.12 `getCanonicalRate` Independent Of Amount

Note **Version 1**

During calculation of the gross asset value of a fund, the exchange rate of each asset to the denomination asset is queried. The `getCanonicalRate` function however is independent of the amount. Exchange rates may depend significantly on the amount to be exchanged. This leads to wrong rates for illiquid assets.

In version 4 of the code `getCanonicalRate` was refactored to `calcCanonicalValue`, this function now returns the value directly instead of the rate. The calculation is still independent of the amount.