PUBLIC

Security Audit

of Melon’s Smart Contracts

March 1, 2019

Produced for

MELONPORT

by

CHAINSECURITY

Table Of Content

Foreword

We first and foremost thank Melonport for giving us the opportunity to audit their smart contracts. This documents outlines our methodology, limitations, and results.

– ChainSecurity

Executive Summary

The Melon Protocol smart contracts have been audited manually by security experts and using automated security tools for Ethereum smart contracts.

The initial audit involved 4 auditors over a period of 2 weeks from January 28th to February 11th, followed by reviewing code updates delivered between February 12th and February 22nd. On request of Melonport ChainSecurity reported critical and high severity issues on an ongoing basis during the audit to facilitate quick remediation.

During the audit process and the code update review process the following issues have been reported:

- Security: two critical, three high, five medium, and nine low severity issues
- Trust: six medium severity issues
- Design: three medium and six low severity isues

Out of these, all critical and high severity issues have been fixed. Most medium and low severity issues have been fixed or addressed.

The project is complex, each fund consists of several contracts which interact with external exchanges and tokens. Security audits of such systems cannot guarantee absence of errors.

Limitations

Security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a secure smart contract. However, auditing allows to discover 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 lack protection against it completely. Some of the issues may affect the entire smart contract application, while some lack protection only in certain areas. We therefore carry out a source code review trying to determine all locations that need to be fixed. Within the customer-determined timeframe, ChainSecurity has performed auditing in order to discover as many vulnerabilities as possible.

Audit Overview

Scope of the Audit

Melonport communicated that the scope of the audit is limited to finding security issues only. Any gas cost optimization should not be reported as they might not consider fixing it.
The scope of the audit is limited to the following source code files. All of these source code files were received on January 29, 2019. The updated code files were received on February 22, 2019:

File	SHA-256 checksum
./version/Version.sol	dc4d12bbcb134dc193c26a462c85b1b3f23a978bb39571850d9b445988c7febc
./version/Registry.sol	b75f34f97e638020b351100ac06d743aae406c930274992fae84e4ab03896ed6
./testing/SelfDestructing.sol	88c8e45efdb7d5ec2724bda184ab8dd699659bed004e87dd451ef55ae3eadb34
./prices/TestingPriceFeed.sol	5677104d5fef3f6b936aae1afa093a18f87703770a6deb684d193c7de55f36e7
./prices/StakingPriceFeed.sol	6fc4625e28e7cee18fee6f5aa6f225968a3cc53cb7db1a50183dac0d11390003
./prices/SimplePriceFeed.sol	a7b11c9fd6280c29052221d02010591411e111cc48b7c7609c3cfe2231f5ee05
./prices/PriceSource.i.sol	4d67c61ea44a0221cb696725f4b86c763473b5ee3557cee1cfc8db47893d5f6f
./prices/OperatorStaking.sol	161fef50cb7e5ef3af79a040c7fff4c4bdcdae5ccf5dda662a48077e6d61a545
./prices/KyberPriceFeed.sol	d3f7d61c7275251582b7b17c6987a750907d001352b9a5ee031218fcbeed7328
./prices/CanonicalRegistrar.sol	aa8ab5d16459cdb6d6632edd9e79257f5ae1e9c9f815e61f4440ba662c895ef5
./prices/CanonicalPriceFeed.sol	4e420e1ae17fbefef839cc82161b18ce25a3a8f6daab783d35798f34887ac3f0
./fund/vault/Vault.sol	a402c28ece18eef0cb3b577e50c045c0cbefdf2561a7091eccc4491ace50cf86
./fund/trading/Trading.sol	0ed6e58bcb0b7e9a4cf9c044512e60bde65bdbd0eb953e24753bde54941c506e
./fund/shares/Shares.sol	defe0ab1c1968240521e8b0bf2d9edd2ac84bf542e723fe536479e688ef46319
./fund/policies/TradingSignatures.sol	1217d9d57d459fcc778cb43b3f903c29c0aae7e47184d3077ebc8cf36a0da9c7
./fund/policies/PolicyManager.sol	4662b39508da78b89d515515acb4b9b40e337617d8a5929fcdac77961a641fdf
./fund/policies/Policy.sol	19a14ad2d5446d9e38e5a4529226c6abbbed10b2609c158cf24f7a2827e9fa9e
./fund/policies/BooleanPolicy.sol	307c5becb8fa6d76cfbbbe13edff1ab2c8a8790339d7bf17567867945d8c7951
./fund/policies/AddressList.sol	c75eaee2b0bd757a55dd5f85732eacece063b4ad4b0b31e40c1620e136071963
./fund/policies/risk-management/PriceTolerance.sol	e23aef22cb449d9c78443befd23f789f9340e99219db9776ff8aacffd8a33b96
./fund/policies/risk-management/MaxPositions.sol	a8d968c24de8ab6429dfe1bc6ea91eb81bb1ef850a27cbfa50b4e0714588f216
./fund/policies/risk-management/MaxConcentration.sol	7064e7a989efb7205230e8163e082a1df04a2deebedbe04a3cf84a6d203116c2
./fund/policies/risk-management/AssetWhitelist.sol	611b55cd62472ca42a48c38d59455a10de6b122f028b3e9cb73a90187954f734
./fund/policies/risk-management/AssetBlacklist.sol	04275f2be7b8dbfa81134800f755c91040be68bb27c2a2936ed00fdb45af3368
./fund/policies/compliance/UserWhitelist.sol	62e3379a01f3e554a9127ccce646caaf174dd3280bcfe2cb5a4647f1f340019c
./fund/participation/Participation.sol	8942126004ef7fd2465860f5788e8013b76f54af210a6a28b72c1e9f05440258
./fund/hub/Spoke.sol	99a4c35a69a0433513744171d9c7a80938609aba70d92bda7284f6bcf47d0a0e
./fund/hub/Hub.sol	90a494ece65a128e324318aaad35d27c0f7270ec0eea2b3576ac17990f95835d
./fund/fees/PerformanceFee.sol	a81f865f22185ff45e6aa666037bb168b988ace2167aaa4c1499ba2edf7983bc
./fund/fees/ManagementFee.sol	0a5819a4db2cfabdfe5ceef62e7af61ff8c2c2717afef0b50c319589522f5b59
./fund/fees/FeeManager.sol	4cdfadedfdde92f72f568514236abc03aa6730771ad7c4bed2861174f9ec5768
./fund/accounting/Accounting.sol	f9a7732ee78fcac8f62a2b71e6272fad9a2fada194fa33e140643fd7dfe0da68
./factory/FundRanking.sol	ec9de5e40bdc68bea0a898aff8f368580b276b4b50af64ce1f26b36ded409697
./factory/FundFactory.sol	eb4b4c09e6d632c90ce1e0591fa79fed6b1cb3e232973053f68a7e6f9ce9457c
./factory/Factory.sol	ff14e73c820c55ca94ac8e122fddc1a55b74eb72d1f96b81536077781ded1c61
./exchanges/ZeroExV2Adapter.sol	766430ea6d91dd7a1b652d35855a0d3bcf48195cb408385741ca3b63e9b6da0e
./exchanges/MatchingMarketAdapter.sol	0790a83c627efb9f96580a4354e14e3ff5984ef7ef196a099bddd1bb769e8d26
./exchanges/MatchingMarketAccessor.sol	a7a1dec4278d3bdf5e459b21bf2a2220a9c9ba368c26b971df67d9e8b16af5c8
./exchanges/KyberAdapter.sol	ea5fa0783ac91d57805e5745e9311babc4452073921d241bfae612b67660d3cc
./exchanges/ExchangeAdapter.sol	8f998924a17f5b628ea5c4ae187033f4f9de042c6d243b5c355a794e4f7b459e
./exchanges/EthfinexAdapter.sol	e0647b58a72d6d8d58b2aee315b505ff8f25557c679d3cf939efc990178b1135
./exchanges/EngineAdapter.sol	789c95652d9e4276acfd43d25dea6b2a5255996282b80c622cde7c43760b8a6f
./engine/Engine.sol	a9d1a91c058a8899fbdb6a7a96ddb2e8a3ca01848d458b4171da1f257910f5cf
./engine/AmguConsumer.sol	6ad6d7ef9cfcbea45c8331ec25754035336cfb78aa1e9bf2fbc3f876250d2956

The corresponding Git commit is: `04b6c0eee2bd2179c0ddc77da6330fb582440632`[1].
The updated code Git commit is: `aafb24f5f3e92c1cace4d76179d706d9e62077e3`[2]

Depth of Audit

The scope of the security audit conducted by ChainSecurity was restricted to:

- Scan the contracts listed above for generic security issues using automated systems and manually inspect the results.
- Manual audit of the contracts listed above for security issues.

Terminology

For the purpose of this audit, we adopt the following terminology. For security vulnerabilities, we specify the *likelihood*, *impact* and *severity* (inspired by the OWASP risk rating methodology[3]).

Likelihood represents the likelihood of a security vulnerability to be encountered or exploited in the wild.

Impact specifies the technical and business related consequences of an exploit.

Severity is derived based on the likelihood and the impact calculated previously.

We categorize the findings into 4 distinct categories, depending on their severities:

- L Low: can be considered as less important
- M Medium: should be fixed
- H High: we strongly suggest to fix it before release
- C Critical: needs to be fixed before release

These severities are derived from the likelihood and the impact using the following table, following a standard approach in risk assessment.

	IMPACT		
LIKELIHOOD	**High**	**Medium**	**Low**
High	C	H	M
Medium	H	M	L
Low	M	L	L

During the audit concerns might arise or tools might flag certain security issues. After careful inspection of the potential security impact, we assign the following labels:

- ✓ No Issue: no security impact
- ✓ Fixed: during the course of the audit process, the issue has been addressed technically
- ✓ Addressed: issue addressed otherwise by improving documentation or further specification
- ✓ Acknowledged: issue is meant to be fixed in the future without immediate changes to the code
- New Issue: This issue has been introduced into the code after the commit used for the initial audit, either as a side-effect of addressing a fix or through unrelated code changes.

Findings that are labelled as either ✓ Fixed or ✓ Addressed are resolved and therefore pose no security threat. Their severity is still listed, but just to give the reader a quick overview what kind of issues were found during the audit.

[1] `https://github.com/melonproject/protocol/tree/04b6c0eee2bd2179c0ddc77da6330fb582440632`
[2] `https://github.com/melonproject/protocol/tree/aafb24f5f3e92c1cace4d76179d706d9e62077e3`
[3] `https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology`

System Overview

Melon intends to provide a decentralized, public and permissionless infrastructure for the secure management of cryptographic tokens inside investment Funds on the Ethereum blockchain.

The protocol consists of two layers:

- The infrastructure-level contracts, managed by the governance system
- Investment Funds, controlled by their respective managers and with their participating investors

Terms

Fund Each Fund is a constellation of smart contracts where each is customized with parameters set by he Fund manager upon creation of the Fund. Anyone may start their own Fund.

Asset Assets, e.g. tokens are parts of the Funds. Assets must be registered (whitelisted) to be able to be included in a Fund.

Fund Manager Creator of the Fund. Manages the Fund, trades to maximize the Funds performance.

Investor Any participant investing in the Fund.

Infrastructure level contracts

These infrastructure related contracts are deployed once for the entire network and are controlled by the governance system.

FundsFactory This contract enables managers to create their own Funds. Factories for each of the Fund components are additionally deployed independently and used during the setup of new Funds.

Exchange Adapters Each supported Exchange has a corresponding Exchange Adapter contract which is deployed once and shared by all Funds. Currently Adapters for `Kyber`, `Ethfinex`, `Oasisdex` and `ZeroEx` are available.

Individual investment Funds

All Funds have their own instance of the following contracts deployed:

Hub The core contract of any Fund. Tracks all other components, which can be thought of as the spokes of the hub. Also provides functionality for setting up the Fund and maintains an access control list for who can call which methods.

Vault This contract stores the assets of the Fund safely.

Shares Shares represent the proportional amount of ownership in a Fund. They can only be created and destroyed by the investors, but cannot be traded.

Participant This contract enables the interaction of Investors with the Fund. It provides functionality to subscribe and redeem, which includes the creation/destruction of shares, calculation of fees and asset token transfers.

Accounting Provides the overall share price for the Fund. Tracks the amount of assets owned by the Fund and computes all relevant metrics. Finally enables the functionality for the fee payment on the Fund.

FeeManager Fees are rewarded to the manager of a Fund. Currently there are two types of fees:

- management fees: calculated based on time of participation in the Fund
- performance fees: determined by share price evolution

A fund manager can claim the accumulated fees in the form of shares. These shares can be redeemed for assets.

Trading A manager is supposed to maximize their Fund's performance by trading. The trading contract provides functionality to interact with multiple exchange adapters. For an individual Funds not all Exchange Adapters may be active.

PolicyManager The PolicyManager contract controls and enforces the compliance policies for the Trading. Policies are individual contracts registered with the PolicyManager contract and define certain rules. To enforce these the PolicyManager validates these and reverts the transaction if any rule is violated.

Policies applying to a Fund are registered by the FundManager and can extended, but not removed. There exists a set of compliance and risk-management policies ready for users, but Fund Managers are free to create and register their own.

The `PriceSource`, `Registry`, `Version`, `Engine` and `mlnToken` contract addresses are set to the default values by the `FundFactory` and are shared by all Funds. The Governance system may change these addresses, this however will only affect new Funds created afterwards.

PriceSource PriceSource where the Fund gets information about asset values. Defined by the governance system, fixed for a Fund once it has been created.

Registry All Funds register with the registry, a contract of the Melon network which keeps a high level overview over the whole system. Furthermore this contracts tracks registered assets and controls the list of available exchanges.

Version The Version contract is the contract creator for all Funds. It defines a specific watermelon protocol version and provides some administrative functionality.

Engine The Engine contract collects fees paid in ETH for the use of the melon protocol. For calculations of usage fees due, Melon introduces AMGU, asset management gas units. Certain functions in the protocol have the `amguPayable` modifier. When interacting with these functions, the caller needs to pay the fees based on the gas consumption during the execution.

The settlement of these fees is payed as follows: The transaction needs to have sufficient ether as `msg.value`. With this, sufficient `MLN` is bought at a premium to market and burned.

MLN The address of the `MLN` Token contract.

Best Practices in Melon's project

Projects of good quality follow best practices. In doing so, they make audits more meaningful, by allowing efforts to be focused on subtle and project-specific issues rather than the fulfillment of general guidelines.

Avoiding code duplication is a good example of a good engineering practice which increases the potential of any security audit.

We now list a few points that should be enforced in any good project that aims to be deployed on the Ethereum blockchain. The corresponding box is ticked when Melon's project fitted the criterion when the audit started.

Hard Requirements

These requirements ensure that the Melon's project can be audited by CHAINSECURITY.

- ☑ The code is provided as a Git repository to allow the review of future code changes.
- ☑ Code duplication is minimal, or justified and documented.
- ☑ Libraries are properly referred to as package dependencies, including the specific version(s) that are compatible with Melon's project. No library file is mixed with Melon's own files.
- ☑ The code compiles with the latest Solidity compiler version. If Melon uses an older version, the reasons are documented.
- ☑ There are no compiler warnings, or warnings are documented.

Soft Requirements

Although these requirements are not as important as the previous ones, they still help to make the audit more valuable to Melon.

- ☑ There are migration scripts.
- ☑ There are tests.
- ☑ The tests are related to the migration scripts and a clear separation is made between the two.
- ☑ The tests are easy to run for CHAINSECURITY, using the documentation provided by Melon.
- ☐ The test coverage is available or can be obtained easily.
- ☑ The output of the build process (including possible flattened files) is not committed to the Git repository.
- ☑ The project only contains audit-related files, or, if not possible, a meaningful separation is made between modules that have to be audited and modules that CHAINSECURITY should assume correct and out of scope.
- ☑ There is no dead code.
- ☑ The code is well documented.
- ☑ The high-level specification is thorough and allow a quick understanding of the project without looking at the code.
- ☐ Both the code documentation and the high-level specification are up to date with respect to the code version CHAINSECURITY audits.
- ☑ There are no getter functions for public variables, or the reason why these getters are in the code is given.
- ☐ Function are grouped together according either to the Solidity guidelines[4], or to their functionality.

[4] https://solidity.readthedocs.io/en/latest/style-guide.html#order-of-functions

Security Issues

This section relates our investigation into securify issues. It is meant to highlight whenever we found specific issues but also mention what vulnerability classes do not appear, if relevant.

Outdated compiler version M ✓ Fixed

Contracts in Melon's code contain a version pragma enforcing compilation with solc compilers version 0.4.21 or more recent. Throughout the Melon code, the exponent operator (**) is used frequently. Solidity version `^0.4.21` is prone to the `ExpExponentCleanup`[5] issue. This issue has been fixed in Solidity version `^0.4.25`.

To avoid accidental compilation with an affected compiler, ChainSecurity recommends to update the Solidity version specified in all the contracts to `pragma solidity ^0.4.25`.

Likelihood: Low
Impact: High

Fixed: Melon has fixed the issue by using Solidity version `^0.4.25` in all contracts files.

Fund Manager can steal ETH when burning MLN Tokens C ✓ Fixed

The Fund Manager can redirect ETH returned by burning MLN Tokens to any address instead of those ETH going to the `Vault` by providing a fake WETH address.

The issue is in the `EngineAdapter` contract. A Fund Manager can perform the following steps to steal ETH.

- Fund Manager calls `callOnExchange()` function present in `Trading` contract with:
  - `exchangeIndex` for the `Engine` contract.
  - `methodSignature` for `takeOrder()` function.
  - `orderAddresses`: [`mlnToken, evilToken, anyRegisteredToken, any...`]
  - `orderValues`: [`mlnAmount, any...`]
  - The rest of the arguments don't matter.
- The `adapterMethodIsAllowed()` function will pass if `EngineAdapter` is enabled.
- The `preValidate()` function call: in theory a policy could stop the evilToken at this point, but nowhere in the documentation does it mention that there are any mandatory policies required for this adapter
- The `assetIsRegistered()` function call checks that `anyRegisteredToken` is registered.
- delegatecall to `takeOrder()` function present in `EngineAdapter` contract.
- Here `orderAddresses[1]` is used for `wethAddress`, which until now has not been verified.
- The MLN Token are burned and ETH is sent to `Trading` (in the `Engine`)
- `Trading` now forwards the ETH to `wethAddress` (using `WETH(wethAddress).deposit.value(ethToReceive)()`).
- Since `wethAddress` was never verified it could also be any Fund Manager controlled contract.
- Now `WETH(wethAddress).transfer(address(vault), ethToReceive)` could then forward the funds to some address that is not the vault (e.g. directly to the Fund Manager).
- None of the remaining calls in the `takeOrder()` function check `wethAddress` address.
- Back in `callOnExchange()` function only `postValidate()` calls remains (once again there is no mention of any required default policies).

Likelihood: High
Impact: High

[5] https://etherscan.io/solcbuginfo

Fixed: Melon solved the problem by explicitly using the native asset address (WETH contract address from `Registry`) to transfer the ETH.

Theoretical integer underflow possible L ✓ Fixed

The contract `PriceTolerance`, function `makeOrder()` contains the code below:

```
int res = int(ratio) - int(_value);
```

PriceTolerance.sol

This unsafe subtraction might underflow in theory, resulting in a different result than expected. OpenZeppelin does provide a SignedSafeMath[6] library. Note that the SignedSafeMath contract is inside a draft folder of the library.
Likelihood: Low
Impact: Medium

Fixed: Melon solved the problem by using the `signedSafeSub()` function from `SignedSafeMath.sol` library code present in OpenZeppelin.

Omitted return value on `send()` call L ✓ Fixed

Inside the `sellAndBurnMln()` function of the `Engine` contract, the following statement is used to send ETH to the `msg.sender`:

```
totalMlnBurned = add(totalMlnBurned, mlnAmount);
msg.sender.send(ethToSend);
mlnToken.burn(mlnAmount);

```

Engine.sol

The `send()` function does return the boolean result indicating whether the transfer was successful. However, this return value is not checked and hence the call to the function `sellAndBurnMln()` would be successful even if `send()` function fails. ChainSecurity recommend to use `transfer()` function instead of `send()`.
Likelihood: Low
Impact: Low

Fixed: Melon solved the problem by using the `transfer()` function instead of `send()`.

Possible integer underflow L ✓ Acknowledged

Inside the `FundFactory` contract, the following function is defined:

```
function getLastFundId() external view returns (uint) { return funds.length -
    1; }
```

FundFactory.sol

When the `funds` array does not have any items present in it. The function call to `getLastFundId()` will return the value after integer underflow.
Also the same function call is being used in `getFundDetails()` function of `FundRanking` contract.

```
uint numberOfFunds = factory.getLastFundId() + 1;
```

FundRanking.sol

Above call would first underflow and then overflow to produce the result `0`.
However, both the functions are `external view` functions, which will not cause any issues in contracts. However, if those functions are called within new contracts, it would return the result after integer underflow or overflow.
Likelihood: Low
Impact: Low

[6] https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/drafts/SignedSafeMath.sol

Acknowledged: Melon acknowledged that the `getlastFundId()` function call they use the integer ID underflow as a convention for there being no funds.

Investor can bypass Performance Fees when redeeming shares M ✓ Fixed

- Attacker calls `callOnExchange()` function present in `Trading` contract with:
  - `exchangeIndex` used for Ethfinex exchange.
  - `methodSignature` for `Ethfinex.withdrawTokens()` function.
  - `orderAddresses`: [`any,any,any,any,any,`someTokenAvailableOnEthfinexButNotRegisteredFor Melonport]
  - `orderValues`: [`0,0,0,0,0,0`]
  - The rest of the arguments don't matter.
  - `adapterMethodIsAllowed()` function call will pass if `EthfinexAdapter` with `withdrawTokens` is enabled
  - The `preValidate()` function call will pass, there is no suitable policy for withdraw.
  - No tokens are checked for registration (since its neither take nor make).
  - Then `delegatecall` to `withdrawTokens()` function in `EthfinexAdapter` contract.
    - No permission check, anyone can call this.
    - Gets the Ethfinex `WrapperToken` (exists if listed on Ethfinex).
    - The balance will be 0 since the fund cannot buy the non-registered token.
    - `WrapperToken.withdraw()` function call will success anyway because the withdraw value is 0.
    - Both `.removeOpenMakeOrder()` and `.returnAssetToVault()` function call will work, the latter simply transfers 0.
    - Asset is added to `Accounting` contract.
  - `postValidate()` function call will pass, there is no suitable policy for withdraw.
- Attacker transfers (`ERC20.transfer`) 1 unit of that token to the `Vault` contract.
  - This will block `Accounting.updateOwnedAssets` from removing it.
- Attacker calls `Participation.redeem()` function.
  - This calls `redeemWithConstraints` with all owned assets (including the non-registered one).
  - Because that one is not registered `PriceSourceInterface(routes.priceSource).hasValidPrices(requestedAssets)` will return false and the fee payments will be skipped.
  - The rest of the function continues as expected.

Likelihood: Medium
Impact: Medium

Fixed: Melon solved the problem by checking that the balance of wrapped token is more than zero in the `Trading` contract. Also the asset is not getting added into the `Accounting` while calling `withdrawTokens()` function of `Ethfinex` contract.

ZeroExV2Adapter allows Fund Manager to circumvent policies H ✓ Fixed

During a `takeOrder()` function call present in the `ZeroExV2Adapter` contract, it extracts the actually used `makerAsset` from `makerAssetData`. Policies and the registry checks in `Trading` however only check the `makerAsset` in `orderAddresses`. By using a different address in `makerAsset` as in `makerAssetData` policies like `PriceTolerance` can be fooled. An evil Fund Manager can then sell assets of the funds (in a trade where the Fund Manager is the market maker) in return for tokens with much less value.

Example: A Fund Manager fools policies into checking a trade of 1 WBTC to 30 WETH (can be any registered tokens), in reality it's for 30 DAI (can be any token that works with 0x)

- Fund Manager creates (himself, not through the fund) a make order on 0x buying 1 WBTC for 30 DAI

- Fund Manager calls `callOnExchange()` function on `Trading` contract
  - With arguments:
    - `exchangeIndex` for 0x exchange.
    - `methodSignature` for `takeOrder()` function.
    - `orderAddresses`: [`any, any, WETH, WBTC`, ...]
    - `orderValues`: [`30WETH, 1BTC, any..., 1BTC`]
    - `identifier` for the Fund Manager created order
    - `makerAssetData`: `assetProxyId` + `DAI`
    - `takerAssetData`: `assetProxyId` + `WBTC`
    - The rest of the arguments don't matter.
  - `adapterMethodIsAllowed()` function call will pass if `ZeroExV2Adapter` is enabled
  - `assetIsRegistered()` function call for `WETH` will pass
  - `preValidate`: policies are only passed the `makerAsset` not `makerAssetData` (e.g. to `PriceTolerance` it looks like a 30 WETH for 1 WBTC trade which seems reasonable)
  - Then `delegatecall` to `takeOrder()` function in `ZeroExV2Adapter`
    - In `approveTakerAsset` takerAsset (1 WBTC) is transferred from Vault and approved for 0x.
    - `executeFill`
      - `takerFee` is processed as normal.
      - `makerAsset` is derived from `makerAssetData` (`DAI` in this case)
      - The trade now takes place in `Exchange(targetExchange).fillOrder`. 0x will also take the `makerAsset` from `order.makerAssetData` and ignore `order.makerAsset` (it is never accessed throughout the contract). Thus the trade is actually 1 WBTC for 30 DAI.
      - The Fund Manager gets 1 WBTC, the fund gets 30 DAI.
      - the preMakerAssetBalance / postMakerAssetBalance will pass because the `makerAsset` variable is DAI
    - `makerAsset` (WETH) is added to assets (and immediately removed in `updateOwnedAssets` if there are none)
    - `makerAsset` is returned to vault (but there is none in asset)
    - the DAI are stuck in `Trading` (they could be manually moved back to the vault by the manager using `returnAssetToVault`)
  - Back in `callOnExchange` only `postValidate` remains.

This can be repeated as many times as needed to drain the fund. (Note: `MaxConcentration` Policy can also not stop this because it only looks at the supplied `makerAsset` which if the fund doesn't have it will remain at 0%).

Likelihood: Medium
Impact: High

Fixed: Melon solved the problem by checking that the passed-in address matched the address from the asset data (bytes).

Omitted return values M ✓ Fixed

In `withdraw()` function of `Vault` contract, to withdraw the ERC20 tokens below function call is made:

```
function withdraw(address token, uint amount) external auth {
    ERC20(token).transfer(msg.sender, amount);
}
```

Vault.sol

However, as per the ERC20 standard interface, this function returns `bool` value as result to let the caller know about the transfer status. To ensure the transfer safety, this function can be enclosed with a `require()` to revert the transaction in case transfer not done successfully.

However, in practice there are two types of ERC20 implementations.

- Some older ERC20 tokens do not provide any return value when functions such as `transfer()` are called. Among, these tokens, there are some popular ones such as OmiseGo[7].
- Token contracts with a correct implementation of the standard, return a `bool` value to let the caller know about the status of the transfer.

This has been a known issue and caused the problem[8] with Decentralized Exchanges (DEXs). ChainSecurity recommends Melon to carefully check whether the `transfer()` function call has succeeded.
Melon may wants to support both types of ERC20 implementations as assets in a Fund.

Also according to the ERC20 token standard following statement is mentioned in the ERC20 specification[9]:
"Callers MUST handle false from returns (bool success). Callers MUST NOT assume that false is never returned!"
Likelihood: Low
Impact: High

Fixed: Melon solved the problem by adding a new contract called `TokenUser` and using the `safeTransfer()` and `safeTransferFrom()` functions it when doing ERC20 token transfers. However, with this fix a new Security issue is introduced which allow anyone to steal funds from `Vault` contract.

Delegate call to exchange adapters M ✓ Acknowledged

As of now Melon supports connecting to following exchanges via their specific exchange adapters:

- Oasis DEX
- 0x Project
- Kyber Network
- EthFinex

In future Melon would add the support for new exchanges via their exchange adapters. As the calls to each of the exchange adapters is done using a `delegatecall`, this could increase security risks.
ChainSecurity wants to highlight the importance of carefully auditing all future adapters as all adapters have full `withdraw` permission for the `Vault`.
Likelihood: Low
Impact: High

Acknowledged: Melon acknowledged that the Melon Council will only register new exchange adapters that have been thoroughly reviewed and audited. No fix required.

Possible implications of `CREATE2` Opcode L ✓ Acknowledged

ChainSecurity wants to raise awareness about the potential implications of the new `CREATE2` opcode proposed in EIP-1014 [10] which will be activated in the upcoming Constantinople hardfork.
In brief, this new opcode changes the previous invariant that code deployed at an address remains constant forever. Code of contracts containing a `SELFDESTRUCT` can be replaced by new code at the same address.
Note that even if there is no `SELFDESTRUCT` present in the bytecode stored at the contract's address, a `SELFDESTRUCT` might still be reached and executed through `CALLCODE` or `DELEGATECALL` operations.
In Melon's code this may have severe consequences at multiple locations. A non-exhaustive list of occurrences where a registered external contract may completely change it's behavior:

- Registered policy's of funds
- `PerformanceFee` and `ManagementFee` contracts
- A registered `denominationAsset`

Likelihood: Low
Impact: Medium

[7]https://etherscan.io/address/0xd26114cd6EE289AccF82350c8d8487fedB8A0C07#code
[8]https://medium.com/coinmonks/missing-return-value-bug-at-least-130-tokens-affected-d67bf08521ca
[9]https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md#methods
[10]https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1014.md

Acknowledged: Melon acknowledged that none of the contracts in the scope of the audit contain the call to `selfdestruct`. In the future, the best would be to avoid including policies, assets, fees or any contracts with a `selfdestruct` opcode.

Dependence on block information L ✓ Acknowledged

There are many contracts which makes use of the special `block.timestamp` field. For example in `PerformanceFee` contract. Although block manipulation is considered hard to perform, a malicious miner is able to move forward block timestamps by up to 900 seconds (15min) compared to the actual time.

ChainSecurity notes that Melon and its users should be aware of this and adhere to the 15 seconds rule[11].

Likelihood: Low
Impact: Low

Acknowledged: Melon acknowledged that they are aware about the implications of this.

Locked asset tokens H ✓ Fixed

The Fund Manager buy the assets and these are added to the list using `getAccounting().addAssetToOwnedAssets(takerAsset)` during `makeOrder()` function call make on the exchange adapters.

However, as a make order does not fill instantly, there might not be any `takerAsset` in `Trading` immediately afterwards. As only the currently still held `makerAsset` is counted in `Trading.updateAndGetQuantityHeldInExchange`, the `Accounting.assetHoldings` of the `takerAsset` is 0.

Anyone can now call `Accounting.updateOwnedAssets()` function prior to the order being filled and thus removing `takerAsset` from the list. When `takerAsset` is "returned" to the `Vault` after a successful trade, it doesn't get added either. It will be in the `Vault` but not count towards `GAV` nor can it be withdrawn during a `redeem`.

This issue applied to OasisDex and 0x exchanges only.

Note: This is not an issue with Ethfinex which has a manual withdrawal procedure where the asset is added.

Likelihood: Medium
Impact: High

Fixed: Melon solved the problem by introducing additional variables which track open make orders against a specific asset. Only once no such make order is left, the asset can be removed.

Unnecessary `payable` fallback function in `EngineAdapter` L ✓ Fixed

`EngineAdapter` has an empty `payable` fallback function. But it is only supposed to being delegatecalled and should never receive any ETH as they would be permanently lost.

The idea was probably to receive the ETH from `Engine(targetExchange).sellAndBurnMln(mlnQuantity)`, however the ETH are received by the fallback function in `Trading` which is already `payable`.

Likelihood: Low
Impact: Low

Fixed: Melon solved the problem by removing the fallback function from the `EngineAdapter`.

Unsafe `withdraw()` function call M ✓ Fixed

The `withdraw()` function present in the `Vault` contract calls the ERC20 `transfer()` function which is unsafe.

As per the ERC20 standard, the `transfer()` function returns the `bool` to let the caller know about the transfer status. However, the call present in the `withdraw()` function does not check successful ERC20 `transfer()` execution.

However, the `Vault.withdraw()` function is called from following contracts:

- Participation

[11]https://consensys.github.io/smart-contract-best-practices/recommendations/#the-15-second-rule

  - redeemWithConstraints()
- KyberAdapter
  - swapNativeAssetToToken()
  - swapTokenToNativeAsset()
  - swapTokenToToken()
- MatchingMarketAdapter
  - makeOrder()
  - takeOrder()
- EngineAdapter
  - takeOrder()
- EthfinexAdapter
  - wrapMakerAsset()
- ZeroExV2Adapter
  - approveTakerAsset()
  - approveMakerAsset()
  - executeFill()

Hence, if above functions are called and `transfer()` function returns `false` (via `withdraw()` function) the call will proceed with rest of the function execution.

CHAINSECURITY recommends to ensure that the ERC20 `transfer()` call must be successful.

Likelihood: Medium

Impact: Medium

Fixed: Melon solved the problem by adding a new contract called `TokenUser` and using the `safeTransfer()` and `safeTransferFrom()` functions it when doing ERC20 token transfers. However, with this fix a new Security issue is introduced which allow anyone to steal funds from `Vault` contract.

Locked Ether in the Trading contract L ✓ Acknowledged

The `Trading` contract has a `payable` fallback function accepting incoming ETH. In all currently available Exchange Adapters this ETH is instantly converted to WETH and no ETH reaches the `Trading` contract.

If somebody by accident transfers ETH to `Trading` directly, the ETH would be stuck there forever.

Likelihood: Low

Impact: Low

Acknowledged: Melon acknowledged that no one is suppose to send ETH to the `Trading` contract. The `payable` fallback function in `Trading` contract is used to receive ETH from the `Engine` contract.

Fund Manager can try to lock-up funds L ✓ Fixed

The Fund Manager can try to lock up the customer assets. This would allow black-mailing and similar issues. In order to do so the Fund Manager creates make orders with high prices that will not be taken and maximum expiration periods of 1 day. These make orders lock up all of the funds. Right after a make order has expired, the Fund Manager creates a new one to consistently keep the funds locked and to prevent investors from calling `redeem`. Therefore, investors only have a tiny window of opportunity to `redeem` their shares.

Likelihood: Low

Impact: Medium

Fixed: Melon solved the problem by introducing a 30-minute asset cooldown, taking effect after a make order, which provides a sufficiently large window for investors to `redeem`.

No check for complete execution of Kyber Orders L ✓ Fixed

Inside the `KyberAdapter` the `takeOrder()` function will trigger direct swaps on the Kyber.Network exchange. When such a swap takes place, the amount of received tokens is returned:

```
        uint actualReceiveAmount = dispatchSwap(
            targetExchange, takerAsset, takerAssetAmount, makerAsset, minRate
        );
```

KyberAdapter.sol

However, this amount is never checked against the expected result, namely `takerAssetAmount`. While the `KyberNetworkProxy` enforces a minimum exchange rate, it does not guarantee a complete filling of the order. Therefore, the order might only be partially filled which would lead to unexpected results and stuck tokens. Note that, the current implementation of the swap always attempts to completely fill an order, but this is not guaranteed for future version. (Contract upgrades can be transparent as the proxy remains the same.)

Likelihood: Low

Impact: Medium

Fixed: Melon solved the problem by checking that `actualReceiveAmount` must be greater than or equal to `makerAssetAmount`.

Trust Issues

This section mentions functionality which is not fixed inside the smart contract and hence requires additional trust into Melon, including in Melon's ability to deal with such powers appropriately.

Untrusted Policy contract injection by Fund Manager M ✓ Acknowledged

Policies for a Fund are to control and enforce the compliance policies for the Trading. Effectively these policies for a Fund are however are added by the Fund Manager, the one who should be restricted by them. The Fund Manager can only add, but not remove registered policies.

While there are already some predefined compliance and risk-management policies, one can easily define own rules by creating own contracts inheriting from `Policy` contract:

```
contract Policy {
    enum Applied { pre, post }
    /*... code commentes present ...*/
    function rule(bytes4 sig, address[5] addresses, uint[3] values, bytes32
        identifier) external view returns (bool);
    function position() external view returns (Applied);
}
```

Policy.sol

with the desired behavior. Any such contract can be registered by the Fund Manager.

Acknowledged: Melon acknowledged that it is a feature of the Melon protocol that the fund manager can customize its risk management profile (can use existing templates or create its own policies contracts). The policies are never granted permission to transfer assets so Melon does not view this as a security concern.

Untrusted Fee contracts injection by Fund Manager M ✓ Fixed

The Fund Manager can pass untrusted contracts like `PerformanceFee` and `ManagementFee` via `Version.beginSetup()` function call. Which would enable the Fund Manager to manipulate the fees charged.

ChainSecurity recommends allowing only trusted contracts for fees charged by the Fund Manager.

Fixed: Melon solved the problem by keeping a curated list of Fee contracts in `Registry` contract and checking those contracts at the time of new Fund creation.

Untrusted contract injection with `denominationAsset` possible M ✓ Fixed

While creating new Fund, the Fund Manager calls the `beginSetup()` function and passes `_denominationAsset` argument as the contract address to get the decimals value via `decimals()` function.

This contract address is further used in the `PerformanceFee` contract to calculate the `initialSharePrice` and `highWaterMark` variables which will affect the `feeAmount()` result.

This contract address is never checked to ensure that it is a trusted contract.

Fixed: Melon solved the problem by checking the `denominationAsset` with the registered assets list.

Requirements for Council to Register New Tokens M ✓ Acknowledged

At the current state of the project, only a limited set of tokens is supported. In the future new tokens will be added to the system.

However, whenever new tokens are added, the Melon council must ensure that at least the following conditions hold for this token. (The list is not exhaustive.)

- The token implements the `decimals()` function and returns a consistent number.
- The token implements a "standard" transfer function. If X tokens are sent from A to B. A loses X tokens and B gains X tokens. This excludes tokens with transfer fees, such as Digix Gold.
- The token is not deflationary over time, e.g. GoldX token. Simply by holding, the token balance will not change.
- It is not an ERC223 token[12]. This is because the current contracts do not contain the required `tokenFallback()` function.
- It is not an ERC1400 token[13]. The security token standards restricts transfers, which could lock up funds.
- It is not an ERC1644 token[14]. This token standard requires that a controller must have power to transfer tokens in case of any unlawful situations. Therefore, trust assumptions do no longer hold.
- Tokens must be listed on the KyberNetwork exchange. If they aren't listed on this exchange, than the price feed cannot be used.
- Tokens must have a deep order book on KyberNetwork. If the order book is not deep enough, the price feed can be manipulated during a short timespan for a relatively low price. Price Feed manipulations allow all kinds of attacks, including inflated performance fees that can be used to steal from investors.
- Tokens with complicated vesting schemes should be avoided, as the vesting can block transfers.
- Tokens cannot be based on other melon funds in order to avoid issues due to recursions.
- Upgradable tokens, e.g. Gemini Dollar, should in general not be listed unless proper assurance has been given about the timings and contents of upgrades.
- Pausable tokens must be approached with care. If a token is frequently paused it can inhibit system performance.

Acknowledged: Melon acknowledged that the Melon Council will ensure above listed requirements for allowing appropriate ERC20 tokens as assets for Fund.

Fund Manager controls many addresses while Trading M

During various trading operations the fund manager has many opportunities to pass control to arbitrary addresses increasing the potential attack surface. This comes mainly from only verifying one of the involved addresses in a trade against the registry (Note: Using a `PriceTolerance` policy ensures that both assets are checked against the registry. Using an `AssetWhitelist` or `AssetBlacklist` is not enough as they only checks one of the addresses, the one that has already been checked against the registry).

Calls to tokens where the **`address`** can be fully controlled by the fund manager include:

- calls to `makerAsset` in `MatchingMarketAdapter.makeOrder()` and `MatchingMarketAdapter.cancelOrder()`
- calls to `takerAsset` in `MatchingMarketAdapter.takeOrder()`
- calls to `makerAsset` in `ZeroExV2Adapter.makeOrder()` and `ZeroExV2Adapter.cancelOrder()`
- calls to `takerAsset` in `ZeroExV2Adapter.takeOrder()`

Calls to tokens which are not checked by the fund but must be listed on the used exchange:

- calls to `makerAsset` in `EthfinexAdapter.makeOrder()`
- calls to all the tokens in `EthfinexAdapter.withdrawTokens()`
- calls to `takerAsset` in `KyberAdapter.takeOrder()`
- calls to the `makerAsset` extracted from `makerAssetData` in `ZeroExV2Adapter.makeOrder()`

[12] https://github.com/Dexaran/ERC223-token-standard
[13] https://github.com/ethereum/eips/issues/1411
[14] https://github.com/ethereum/EIPs/issues/1644

- calls to the `makerAsset` extracted from `makerAssetData` and the `takerAsset` extracted from `takerAssetData` in `ZeroExV2Adapter.takeOrder()`

Furthermore because the fund manager can add policies at any time control flow can also be passed to custom code at:

- `PolicyManager.preValidate()`
- `PolicyManager.postValidate()`

If the Fund Manager is a contract any of these calls could attempt to reenter into the `ExchangeAdapter` function via `callOnExchange`. While some of the currently used DEXes have reentrancy protection, this extended control of the fund manager allows multiple interactions with different DEXes. Therefore existing policies could be violated. As an example, this means multiple trades could take place between `preValidate` and `postValidate`. Therefore it is important that `preValidate` policies don't try to make any predictions about how the state would change. Any such check must be made via a `post` policy.

ChainSecurity recommends to allow only trusted addresses to reduce the attack vector by the Fund Manager.

Fixed: Melon fixed the issue by checking the different `address` variables passed to these function calls with the `Registry.assetIsRegistered()` function.

However, there are still few functions which are not checking the passed in as `address` variables. These functions are:

- `MatchingMarketAdapter.cancelOrder()`
- `ZeroExV2Adapter.cancelOrder()`
- `EthfinexAdapter.withdrawTokens()`

As the function `Trading.callOnExchange()` checks asset addresses only when the `methodSelector` is either `hex'79705be7'` (makeOrder) or `hex'e51be6e8'`(takeOrder). Hence the delegate call to the above listed functions would no ensure that the asset address is registered.

Contract migration / upgrading M ✓ Acknowledged

The design of Melon allows multiple central components, such as `Version`, `Registry` and Exchange adapters, to be migrated. In case of a migration the new smart contracts have to be carefully evaluated to make sure that no functionality breaks, no vulnerabilities are contained and no additional permissions are granted. In practice, every user should have the technical skills to review the proposed upgrades independently, without relying on third-party explanations. Given the migration features of Melon's project, there is no real bound on how complex future migrations will be, which in turn implies that the minimum required technical skills of users is unknown as of now.

Acknowledged: Melon acknowledged that they will make sure about the concerns raised in this issue while doing contract migration / upgrade.

Design Issues

This section lists general recommendations about the design and style of Melon's project. They highlight possible ways for Melon to further improve the code.

Wrong event emitted L ✓ Fixed

The `UserWhitelist` contract provides whitelisting functionality. The contract defines two events:

```
    event ListAddition(address indexed who);
    event ListRemoval(address indexed who);
```

UserWhitelist.sol

The `addToWhitelist()` and `removeFromWhitelist()` function both trigger the `ListAddition` event.

```
    function removeFromWhitelist(address _who) public auth {
        whitelisted[_who] = false;
        emit ListAddition(_who);
    }
```

UserWhitelist.sol

However, `removeFromWhitelist()` function should trigger the `ListRemoval` event instead, which is currently never used.

Fixed: Melon solved the problem by emitting the `ListRemoval` event from the `removeFromWhitelist()` function.

Division before multiplication leads to loss of precision M ✓ Fixed ✓ Acknowledged

Division before multiplication may lead to loss of precision:

```
uint askRate = 10 ** (KYBER_PRECISION * 2) / bidRateOfReversePair;
// Check the the spread and average the price on both sides
uint spreadFromKyber = mul(
    sub(askRate, bidRate),
    10 ** uint(KYBER_PRECISION)
) / bidRate;
uint averagePriceFromKyber = add(bidRate, askRate) / 2;
kyberPrice = mul(
    averagePriceFromKyber,
    10 ** uint(ERC20Clone(_quoteAsset).decimals()) // use original quote
        decimals (not defined on mask)
) / 10 ** uint(KYBER_PRECISION);
```

KyberPriceFeed.sol

The variable `askRate` is calculated including a division which truncates the rest. This value is later used in a another multiplication.

The variable `averagePriceFromKyber`'s calculation includes a division by to which truncates the result for odd numbers being divided. This variable is later used in a multiplication.

Multiplications before division also present in `PerformanceFee.sol`:

```
uint sharePriceGain = sub(gavPerShare, highWaterMark[msg.sender]);
uint totalGain = mul(sharePriceGain, shares.totalSupply()) / DIVISOR;
uint feeInAsset = mul(totalGain, performanceFeeRate[msg.sender]) / DIVISOR;
uint preDilutionFee = mul(shares.totalSupply(), feeInAsset) / gav;
feeInShares =
    mul(preDilutionFee, shares.totalSupply()) /
    sub(shares.totalSupply(), preDilutionFee);
```

PerformanceFee.sol

These calculations leads to rounding errors as the Ethereum Virtual Machine only works with unsigned integers and thus should be avoided. In order to reduce precision loss, multiplications should precede divisions.

Fixed: Melon fixed the problem by multiplying the required fields first then performing division operation. However, Melon only applied this fix in `KyberPriceFeed.sol`.

Acknowledged: For the `PerformanceFee.sol` Melon acknowledged that the loss of precision during the calculation of the fees. Therefore, for readability purposes no fix will be applied.

Function state mutability can be restricted to `view` L ✓ Fixed

The following functions present in the various contracts are not modifying the state of the contract. For these functions state mutability can be restricted to `view`.

```
    function isInstance(address _child) public returns (bool) {
        return childExists[_child];
    }
```

Factory.sol

```
    function assetMethodIsAllowed(address _asset, bytes4 _sig)
        external
        returns (bool)
```

Registry.sol

```
    function adapterMethodIsAllowed(
        address _adapter, bytes4 _sig
    )
        external
        returns (bool)
```

Registry.sol

CHAINSECURITY recommend using `view` function modifier to allow checking the variable state using Solidity message calls. This would also benefit web3.js clients to read the state of the corresponding variables.

Fixed: Melon solved the problem by using `view` function modifier for the above listed functions.

Same Fund name stops `completeSetup()` M ✓ Fixed

Same fund name is not allowed, however it has been checked at the last step in function `completeSetup()` which would incur loss of funds and fees paid for AMGU(Asset Management Gas Unit). A Fund Manager follows the steps:

- A Fund Manager beings creating a new Fund by following calling `FundFactory.beginSetup()` (on deployed `Version` contract).
- He further calls the other functions present in the `FundFactory` contract to create the requires `Spokes` around the Fund.
- During all the successive function calls Manager is paying AMGU fees and creating require contracts.
- At the last step the Manager calls `completeSetup()` function which registers the Fund in the `Register` contract by calling `registerFund()` function which intern checks the existing fund name.
- In case the Fund name already exists, the Fund Manager cannot gets his fund registered.

Hence the `completeSetup()` function call would always fail for his Fund. Thus, loss of Manager's ETH paid for transactions to setup the Fund and AMGU fees.

However, An attacker also do front running attack to reserver the Fund's name as the Fund name is known in the `beginSetup()` function and only gets registered in `completeSetup()` function. Although it would involve some cost for an attacker to do so.

Fixed: Melon solved the problem by registering the Fund name at the beginning of a new Fund creation process, during the `Version.beginSetup()` function call. Which intern calls `Registry.reserveFundName()` function. However, fix for this issue introduces a new reentrancy issue with the Fund name creation, described under Security issues.

Fields in Struct Asset can be reordered to save storage space L

Storage in the state of Ethereum is organized in storage slots of 256 bits. Subsequent fields of a struct are stored in the same 256bit storage slot if there is enough space remaining. Solidity is still unable to optimize the arrangement of the fields of the struct and if a field does not fit anymore it just uses another storage slot.

```
    struct Asset {
        bool exists;
        string name;
        string symbol;
        uint decimals;
        string url;
        uint reserveMin;
        uint[] standards;
        bytes4[] sigs;
    }
```

Registry.sol

Moving `bytes4[] sigs` into the same field as `bool exist` save one whole storage slot at a cost of 20'000 gas.

No writes to mapping M ✓ Fixed

In the `Participation` contract there is a mapping variable `hasInvested` present. This variable is used in the contract but never written to:

There is a check in `executeRequestFor()` function like below:

```
if (!hasInvested[request.investmentAsset]) {
    historicalInvestors.push(requestOwner);
}
```

Participation.sol

The statement `!hasInvested[request.investmentAsset]` will always return true. Either the `if` block is unnecessary or more likely it should be used somewhere.

ChainSecurity recommends defining the clear usage of the variable otherwise remove it.

Fixed: Melon solved the problem by writing to the variable `hasInvested` and flagging appropriately.

Unneccessary and wrong check in `Participation.requestInvestment()` L ✓ Fixed

The check for the incentive in `Participation.requestInvestment()` function is redundant.

```
require(
    msg.value >= Registry(routes.registry).incentive(),
    "Incorrect␣incentive␣amount"
);
```

Participation.sol

This function has the `amguPayable(true)` modifier which already checks the following via `AmguConsumer` contract:

```
require(
    msg.value >= add(ethToPay, incentiveAmount),
    "Insufficent␣AMGU␣and/or␣incentive"
);
```

AmguConsumer.sol

where `incentiveAmount()` is the value returned by `Registry(routes.registry).incentive()`, the getter function of a trusted contract. Note that the duplicate check only ensures `msg.value > incentive` and misses the ETH needed for AMGU.

Fixed: Melon solved the problem by removing the check from the `requestInvestment()` function.

Unused import L ✓ Fixed

Contract `Registry` imports `TokenUser` contract. However, this is not used anywhere.

Fixed: Melon solved the problem by removing the unused import statement.

Assumed order in Fee array not checked L ✓ Fixed

The function `FeeManager.rewardManagementFee()` assumes that the first item in the `fees` array is always the `ManagementFee` contract.

```
function rewardManagementFee() public {
        if (fees.length >= 1) _rewardFee(fees[0]);
}
```

FeeManager.sol

The same for the `PerformanceFee` as well. It is assumed to be at the second position in the array:

```
function performanceFeeAmount() public view returns (uint) {
        if (fees.length < 2) return 0;
        return fees[1].feeAmount();
}
```

FeeManager.sol

However, this order is not checked when the `fees` array is sent to the constructor of the `FeeManager` contract.

ChainSecurity recommend that in order to prevent deployment errors an additional check could be performed whether the fees have been supplied in the correct order.

Fixed: Melon solved the problem by exposing the `identifier()` function for each Fee type which is checked in `FeeManager` constructor to ensure that the order of the Fee contracts in `fees` array is maintained.

Recommendations / Suggestions

✓ The code has many compilation warnings present. There are many occurrences of below listed warning messages which Melon might consider fixing before mainnet deployment.

- Warning: Defining constructors as functions with the same name as the contract is deprecated. Use "constructor(...) ... " instead.
- Warning: Experimental features are turned on. Do not use experimental features on live deployments.
- Warning: "sha3" has been deprecated in favour of "keccak256"

✓ The constructor of the `PriceTolerance` contract contains:

```
    require(
        _tolerancePercent >= 0 && _tolerancePercent <= 100,
        "Tolerance range is 0% - 100%"
    );
```

PriceTolerance.sol

An `uint` is always `>=0`, check for this is wasting gas.

✓ A comment in present in the contract `StandardToken`:

```
    function _burnFrom(address _account, uint256 _amount) internal {
        require(_amount <= allowed[_account][msg.sender]);

        // Should https://github.com/OpenZeppelin/zeppelin-solidity/issues
            /707 be accepted,
        // this function needs to emit an event with the updated approval.
        allowed[_account][msg.sender] = allowed[_account][msg.sender].sub(
            _amount);
        _burn(_account, _amount);
    }
```

StandardToken.sol

This PR[15] has been accepted and `burnFrom()` and `transferFrom()` functions now emits an `Approval` event in the reference implementation of `OpenZeppelin`. Melon is advised to address this and decide how to proceed. The chosen solution should be consistent in `burnFrom()` and `transferFrom()` functions.

✓ In the `FundFactory.sol` contract an event `NewFund` is declared like below:

```
    event NewFund(
        address manager,
        address hub,
        address[12] routes
    );
```

FundFactory.sol

Indexing these event fields (`address manager` and `address hub`) enables efficient filtering for these addresses.

✓ There is no protection for contract `Engine` to know whether the contract is initialized or not. The contract will become operational after deployment, however most calls will just fail as the registry is not set yet. As the initialization (setting the registry) is separated from the constructor.

[15] https://github.com/OpenZeppelin/openzeppelin-solidity/pull/1524

- ✓ In the FundFactory constructor the **address** _registry parameter is passed and remain unused.

  This variable is already set by Version contract constructor.

- ✓ PerformanceFee can keep track of the performance fee for many different funds, but when a new High-WaterMark is reached it fires an HighWaterMarkUpdate event with the new HighWaterMark as the only parameter. It might be useful to add the address of the related fund in the event. Otherwise an observer to the events cannot tell for which fund the HighWaterMarkUpdate event is triggered.

- ✓ In the KyberPriceFeed.sol contract getPrices() function contain a code snippet like below:

```
        for (uint i; i < _assets.length; i++) {
            uint price;
            uint timestamp;
            (price, timestamp) = getPrice(_assets[i]);
            prices[i] = price;
            timestamps[i] = timestamp;
        }
```

  KyberPriceFeed.sol

  The local variables price and timestamp are unnecessary. The values can be stores without using temporary storage variables.

- ✓ In the PerformanceFee contract there is a mapping variable initialSharePrice which is written in initializeForUser() function call. However, the variable is never used in the contract.

- ☐ Battle-tested libraries should be used as much as possible, since they are less likely to contain bugs than custom code. There are different ways to use library code however, and copy-pasting code without further justification is not a good one.

- ✓ The document says the following:

  “he Vault provides extra security around ERC20 deposit and withdraw functionality, ... by ensuring that only the owner can withdraw from the vault when the vault’s locked state is false”

  However, the locked variable and onlyUnlocked modifier are not present in the code.

Post-audit comment: Melon has fixed some of the issues above and is aware of all the implications of those points which were not addressed. Given this awareness, Melon has to perform no more code changes with regards to these recommendations.

Appendix

In this section, ChainSecurity lists the new issues discovered during the audit phase which have note been present in the originally audited code.

New Issue Anyone can steal funds from Vault C ✓ Fixed

Melon has added new `TokenUser` contract to ensure the safety of ERC20 tokens transfers. The `TokenUser` contract has two functions `safeTransfer()` and `safeTransferFrom()` their visibility is not specified which will be set to `public` by default from Solidity compiler. This introduces a new issue where an attacker can steal any ERC20 tokens from all the contracts inherited from `TokenUser`.

Following are the contracts which inherits `TokenUser` contract.

- `Vault`
- `EngineAdapter`
- `Participation`
- `Trading`

Likelihood: High
Impact: High

Fixed: Melon solved the problem by setting the visibility of the functions `safeTransfer()` and `safeTransferFrom()` to `internal`.

New Issue Front running registration of Fund names blocks creation of new Funds H ✓ Fixed

Funds can only be created with a name that hasn't been used before in a fund connected to the registry used. Normally, fund names are registered upon execution of the `beginSetup()` function of the `FundFactory` contract, which calls `reserveFundName()` with the intended name. This function will only return successfully if the name is either not in use or has been registered by `msg.sender` previously.

`reserveFundName(address _owner, string _name)` of the `Registry` contract can be called by anyone with no restrictions. This allows an attacker to inhibit the creation of any new Fund:

By observing transactions sent to the Ethereum network he detects any transactions executing the function `reserveFundName()`, either by directly calling this function or if called inside another function like `beginSetup()`. Upon detection of such a transaction, the attacker immediately initiates a transaction to register this name on his own, using a high gas price for his transaction. This transaction will then effectively front-run the original transaction as miners give priority to transaction paying a higher gas price. Additionally the attacker may try to increase the propagation speed of his transaction by spreading it to the network over multiple nodes distributed worldwide, to increase his chances. After the attacker's transaction was mined first, the original transaction of the user will revert.

If done continuously by the attacker and if the front running is successful all the time, this effectively inhibits the creation of new funds.

Likelihood: Medium
Impact: High

Fixed: Melon solved the problem by allowing only `Version` contract to call `reserveFundName()` function. However, the front running is still possible but the scope of doing the attack is shortened as it was not the case before this fix.

Disclaimer

UPON REQUEST BY MELON, CHAINSECURITY LTD. AGREES MAKING THIS AUDIT REPORT PUBLIC. THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND, AND CHAINSECURITY LTD. DISCLAIMS ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT. COPYRIGHT OF THIS REPORT REMAINS WITH CHAINSECURITY LTD..