Security Review Report
NM-0072 Arcadia

NETHERMIND

(Mar 2, 2023)

Contents

1 Executive Summary 2

2 Audited Files 3
2.1 Vaults . . . 3
2.2 Lending . . . 3

3 Summary of Issues 4

4 System Overview 5
4.1 `Arcadia Vaults` . . . 5
4.1.1 `Proxy and Factory` . . . 5
4.1.2 `Vault` . . . 6
4.1.3 `MainRegistry` . . . 6
4.1.4 `Liquidator` . . . 6
4.1.5 `OracleHub` . . . 7
4.2 `Arcadia Lending` . . . 7
4.2.1 `Tranche` . . . 7
4.2.2 `LendingPool` . . . 8

5 Risk Rating Methodology 9

6 Issues 10
6.1 [Critical] Unused collateral can be stolen from vaults through the lending pool . . . 10
6.2 [Critical] Vault's `baseCurrency` may differ from the one set by the TrustedCreditor . . . 10
6.3 [Critical] Vaults can borrow from a lending pool different from their trusted creditor . . . 11
6.4 [High] Privileged vault roles not reset upon ownership change . . . 11
6.5 [Medium] Incorrect calculation of `openDebt` during Vault liquidation . . . 12
6.6 [Medium] Pricing module exposure can be manipulated for `WETH` token . . . 13
6.7 [Medium] Vault deployment susceptible to Denial of Service Attacks . . . 13
6.8 [Medium] `minPriceMultiplier` is not scaled to 20 decimals in `_calcPriceOfVault(...)` . . . 14
6.9 [Low] Asset can be added with an inactive oracle or with no oracles . . . 15
6.10 [Low] Asset can be added with an incorrect oracle sequence . . . 15
6.11 [Low] Liquidator contract owner can manipulate live auction pricing . . . 16
6.12 [Low] Oracle's data is not checked to be up-to-date . . . 16
6.13 [Low] Rounding issues in DebtToken shares calculation can steal funds from LendingPool . . . 17
6.14 [Low] The second-most junior tranche can avoid loss by front-running the function `settleLiquidation(...)` . . . 17
6.15 [Low] Vault asset limit can be bypassed with `vaultManagementAction(...)` . . . 18
6.16 [Low] Vault beneficiaries indirectly obtain the asset manager role . . . 18
6.17 [Undetermined] ERC721 pricing modules exposure manipulation . . . 18
6.18 [Info] Blocking the latest vault version does not change `latestVaultVersion` . . . 19
6.19 [Info] Closing margin account doesn't reset `liquidator` . . . 19
6.20 [Info] Function `changeGuardian(...)` does not emit the `oldGuardian` . . . 20
6.21 [Info] Functions will default to USD if an invalid `baseCurrency` is used . . . 20
6.22 [Info] Handling of ERC721 Assets May Be Incorrect in Action Handler . . . 20
6.23 [Info] Initial curve for pricing vaults is not decreasing . . . 21
6.24 [Info] Some common ERC20 tokens can't be deposited . . . 21
6.25 [Info] Unnecessary function `onERC721Received(...)` in factory contract . . . 21
6.26 [Info] `assetIds` length may differ from `amounts` . . . 21
6.27 [Best Practices] Possible optimization in `getListOfValuesPerAsset(...)` . . . 22
6.28 [Best Practices] Storage load optimizations . . . 22

7 Documentation Evaluation 23
7.1 Documentation inconsistencies . . . 23
7.1.1 OracleHub documentation is contradictory . . . 23
7.1.2 Incorrect inline documentation for `_IMPLEMENTATION_SLOT` . . . 23

8 Test Suite Evaluation 24
8.1 Contracts Compilation Output . . . 24
8.1.1 Arcadia Vault . . . 24
8.1.2 Arcadia Lending . . . 24
8.2 Tests Output . . . 24
8.2.1 Arcadia Vault . . . 24
8.2.2 Arcadia Lending . . . 27
8.3 Code Coverage . . . 28
8.3.1 Arcadia Vault . . . 28
8.3.2 Arcadia Lending . . . 28
8.4 Slither . . . 28

9 About Nethermind 29

1 Executive Summary

This document presents the security review performed by Nethermind on the Arcadia Finance contracts. Arcadia Finance is an open-source lending and borrowing protocol allowing users to access decentralized exchanges with borrowed funds. The audit focuses on two of their products: Arcadia Vaults and Arcadia Lending.

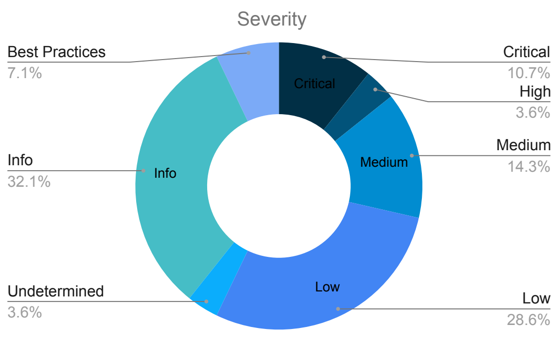
Arcadia Vaults are the core of **Arcadia Finance's** product suite and are responsible for locking, pricing, and managing collateral. They are user-controlled vaults enabling the on-chain pricing of any combination of different types of assets in one single base currency. These vaults are non-custodial and function as decentralized and composable margin accounts, allowing individuals, DAOs, and other protocols to deposit and manage various assets as collateral. They also provide access to on-chain native leverage without requiring the creation of folded positions (repeatedly borrowing and depositing as much as possible) in lending protocols.

Arcadia Lending is the first financial product developed on top of the **Arcadia Vaults**. It is a non-custodial peer-to-contract lending protocol that allows users to borrow against a combination of assets. With **Arcadia Vaults** and **Arcadia Lending**, borrowers have the opportunity to borrow against their entire portfolio with just one position. Additionally, **Arcadia Lending** improves lenders' experience by offering multiple options to provide liquidity depending on their risk appetite.

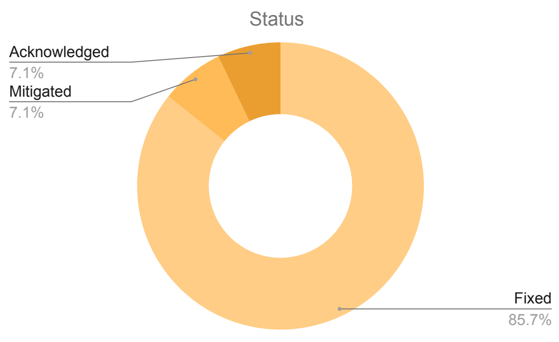
The audited code consists of 2,329 lines of Solidity. Of these lines, 1600 correspond to **Arcadia Vault** with code coverage of 92.75%. The other 729 corresponds to **Arcadia Lending** with code coverage of 100%. Overall, the audited code is of high quality and shows a strong understanding of Solidity's best practices. The team provided documentation presenting the main use cases and protocol mechanisms. This documentation was supported by multiple walkthroughs of the code and continuous communication with the team. **The audit was performed using** (a) manual analysis of the codebase, (b) automated analysis tools, and (c) simulation of the smart contracts.

Along this document, we report 28 points of attention, where three are classified as `Critical`, one is classified as `High`, four are classified as `Medium`, eight are classified as `Low`, eleven are classified as `Info` or `Best Practice`, and one with an `Undetermined` severity. Some of the most relevant issues are related to the lack of checks when interacting with other trusted entities. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, coverage, and automated tests. Section 9 concludes the document.



(a)



(b)

Fig. 1: Distribution of issues: Critical (3), **High** (1), **Medium** (4), **Low** (8), **Undetermined** (1), **Informational** (9), **Best Practices** (2). **Distribution of status: Fixed** (24), **Acknowledged** (2), **Mitigated** (2), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Feb. 27, 2023
Response from Client	Mar. 1, 2023
Final Report	Mar. 2, 2023
Methods	Manual Review, Automated Analysis
Repository	Vaults, Lending Protocol
Commit Hash (Initial Audit - Vault)	7baf91c039974d592df95f8b6c92d4ba10a11ea6
Commit Hash (Initial Audit - Lending)	88a3bd12f02beaa1548e8ddaf53a20d29d4b7880
Commit Hash (Reaudit - Vault)	0c2dc7b65e618c1036ecadb40ad3ee00962c2415
Commit Hash (Reaudit - Lending)	476620fab35d1ba993f0a92820d4a9197debb73c
Documentation	README.md - Vaults, README.md - Lending, Protocol docs
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

2.1 Vaults

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/MainRegistry.sol	224	175	78.1%	51	450
2	src/Factory.sol	140	126	90.0%	44	310
3	src/OracleHub.sol	97	78	80.4%	21	196
4	src/Vault.sol	468	392	83.8%	108	968
5	src/Liquidator.sol	151	158	104.6%	43	352
6	src/RiskModule.sol	39	21	53.8%	4	64
7	src/Proxy.sol	35	37	105.7%	13	85
8	src/actions/MultiCall.sol	31	19	61.3%	10	60
9	src/actions/ActionBase.sol	9	11	122.2%	4	24
10	src/actions/utils/ActionData.sol	8	6	75.0%	1	15
11	src/PricingModules/StandardERC20PricingModule.sol	58	62	106.9%	18	138
12	src/PricingModules/AbstractPricingModule.sol	112	84	75.0%	29	225
13	src/PricingModules/interfaces/IStandardERC20PricingModule.sol	4	12	300.0%	1	17
14	src/PricingModules/interfaces/IOraclesHub.sol	5	17	340.0%	2	24
15	src/PricingModules/interfaces/IMainRegistry.sol	5	14	280.0%	2	21
16	src/interfaces/IERC721.sol	5	1	20.0%	2	8
17	src/interfaces/IFactory.sol	6	22	366.7%	3	31
18	src/interfaces/ITrustedCreditor.sol	5	18	360.0%	2	25
19	src/interfaces/IPricingModule.sol	13	32	246.2%	5	50
20	src/interfaces/IVault.sol	9	33	366.7%	5	47
21	src/interfaces/IERC20.sol	7	1	14.3%	4	12
22	src/interfaces/IActionBase.sol	5	11	220.0%	2	18
23	src/interfaces/ILendingPool.sol	11	10	90.9%	1	22
24	src/interfaces/IChainLinkData.sol	16	4	25.0%	7	27
25	src/interfaces/IERC1155.sol	5	1	20.0%	2	8
26	src/interfaces/IERC4626.sol	6	1	16.7%	3	10
27	src/interfaces/IMainRegistry.sol	36	64	177.8%	10	110
28	src/security/MainRegistryGuardian.sol	36	48	133.3%	12	96
29	src/security/BaseGuardian.sol	18	55	305.6%	12	85
30	src/security/FactoryGuardian.sol	36	48	133.3%	12	96
	Total	**1600**	**1561**	**97.6%**	**433**	**3594**

2.2 Lending

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/InterestRateModule.sol	32	27	84.4%	6	65
2	src/TrustedCreditor.sol	13	32	246.2%	6	51
3	src/Tranche.sol	133	95	71.4%	45	273
4	src/DebtToken.sol	54	79	146.3%	23	156
5	src/LendingPool.sol	376	334	88.8%	102	812
6	src/libraries/DataTypes.sol	9	6	66.7%	1	16
7	src/interfaces/IFactory.sol	5	16	320.0%	2	23
8	src/interfaces/IVault.sol	6	21	350.0%	3	30
9	src/interfaces/ITranche.sol	5	12	240.0%	2	19
10	src/interfaces/ILendingPool.sol	8	26	325.0%	5	39
11	src/interfaces/ILiquidator.sol	4	12	300.0%	1	17
12	src/security/Guardian.sol	84	85	101.2%	18	187
	Total	**729**	**745**	**102.2%**	**214**	**1688**

3 Summary of Issues

	Finding	Severity	Update
1	Unused collateral can be stolen from vaults through the lending pool	Critical	Fixed
2	Vault's `baseCurrency` may differ from the one set by the TrustedCreditor	Critical	Fixed
3	Vaults can borrow from a lending pool different from their trusted creditor	Critical	Fixed
4	Privileged vault roles not reset upon ownership change	High	Fixed
5	Incorrect calculation of `openDebt` during Vault liquidation	Medium	Fixed
6	Pricing module exposure can be manipulated for `WETH` token	Medium	Fixed
7	Vault deployment susceptible to Denial of Service Attacks	Medium	Fixed
8	`minPriceMultiplier` is not scaled to 20 decimals in `_calcPriceOfVault(...)`	Medium	Fixed
9	Asset can be added with an inactive oracle or with no oracles	Low	Fixed
10	Asset can be added with an incorrect oracle sequence	Low	Fixed
11	Liquidator contract owner can manipulate live auction pricing	Low	Fixed
12	Oracle's data is not checked to be up-to-date	Low	Mitigated
13	Rounding issues in DebtToken shares calculation can steal funds from LendingPool	Low	Fixed
14	The second-most junior tranche can avoid loss by front-running the function `settleLiquidation(...)`	Low	Acknowledged
15	Vault asset limit can be bypassed with `vaultManagementAction(...)`	Low	Fixed
16	Vault beneficiaries indirectly obtain the asset manager role	Low	Fixed
17	ERC721 pricing modules exposure manipulation	Undetermined	Fixed
18	Blocking the latest vault version does not change `latestVaultVersion`	Info	Mitigated
19	Closing margin account doesn't reset `liquidator`	Info	Fixed
20	Function `changeGuardian(...)` does not emit the `oldGuardian`	Info	Fixed
21	Functions will default to USD if an invalid `baseCurrency` is used	Info	Fixed
22	Handling of ERC721 Assets May Be Incorrect in Action Handler	Info	Fixed
23	Initial curve for pricing vaults is not decreasing	Info	Acknowledged
24	Some common ERC20 tokens can't be deposited	Info	Fixed
25	Unnecessary function `onERC721Received(...)` in factory contract	Info	Fixed
26	`assetIds` length may differ from `amounts`	Info	Fixed
27	Possible optimization in `getListOfValuesPerAsset(...)`	Best Practices	Fixed
28	Storage load optimizations	Best Practices	Fixed

4 System Overview

The audit consists of two phases: a) Arcadia Vaults and b) Arcadia Lending. The Arcadia Vaults are the core of Arcadia Finance and are responsible for locking, pricing, managing, and liquidating collateral. The scope of this part consists of 30 contracts. Arcadia Lending is a protocol where users can borrow against a combination of assets. The scope of the lending part embraces 12 contracts.

4.1 Arcadia Vaults

Fig.2 presents the core contracts in Arcadia Vaults and their dependencies. In summary, `Proxy`, `Factory`, `Vault`, `MainRegistry`, `Liquidator`, and `OracleHub` are the contracts that implement key features provided by Arcadia Vaults. `LendingPool` is part of Arcadia Lending.

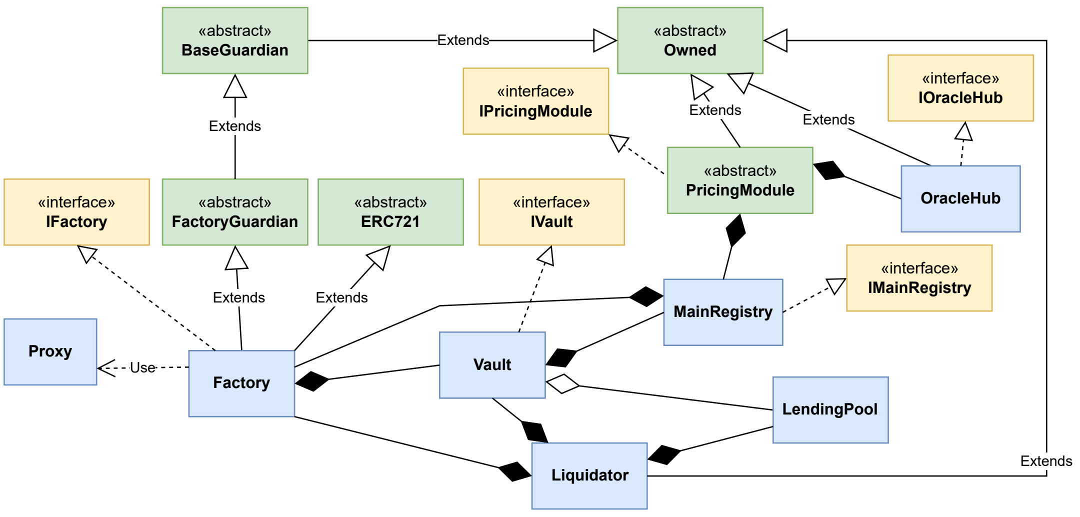


Fig. 2: Arcadia Vault - Structural Diagram of the Contracts

4.1.1 Proxy and Factory

The `Factory` extends the abstract contracts `FactoryGuardian` and `ERC721`. Arcadia Vaults themselves are according to the ERC-721 standard and can thus be represented as a single asset. A vault is user-controlled and it is deployed through the proxy factory by the user calling the function `createVault(...)`. Then, the user becomes the single owner of the deployed vault. The `Factory` manages all vaults with their respective version information. This contract has the following public functions. For simplicity reasons, we do not list functions in the contract `ERC721` since it is a well-known open standard for unique tokens. We also omitted functions implemented in `Owned`.

- `createVault(...)`
- `upgradeVaultVersion(...)`
- `safeTransferFrom(...)`
- `transferFrom(...)`
- `setNewVaultInfo(...)`
- `blockVaultVersion(...)`
- `liquidate(...)`
- `setBaseURI(...)`
- `changeGuardian(...)`
- `pause(...)`
- `unpause(...)`

In addition to these functions, the contract provides 6 (six) `getters` which are not described here.

4.1.2 `Vault`

A deployed vault is linked to a specific vault logic contract that includes features from which the vault owner can benefit, such as flash withdrawals and active collateral management. This vault logic is upgradeable, but the vault owner can decide if or when they want to upgrade to the new version. Users can deposit assets (ERC20, ERC721, ERC1155, ...) in their respective vaults. The vault denominates all the pooled assets into one `baseCurrency` (one unit of account, e.g., USD or ETH). Users can use the single-denominated value of all their assets to take margin (take credit line, financing for leverage...). This contract has the following public functions:

- `initialize(...)`
- `upgradeVault(...)`
- `upgradeHook(...)`
- `transferOwnership(...)`
- `setBaseCurrency(...)`
- `openTrustedMarginAccount(...)`
- `closeTrustedMarginAccount(...)`
- `liquidateVault(...)`
- `setAssetManager(...)`
- `vaultManagementAction(...)`
- `deposit(...)`
- `withdraw(...)`
- `skim(...)`

In addition to these functions, the contract also provides 10 (ten) `getters` which are not described here.

4.1.3 `MainRegistry`

The `MainRegistry` is a core part of the protocol. It contains all the information related to valid base currencies and their respective pricing modules, and it is the entity able to calculate the value stored in a Vault. End-users should not directly interact with the contract `MainRegistry`, only Vaults, Pricing Modules, or the contract owner. The public functions implemented in the contract are listed below:

- `setAllowedAction(...)`
- `addBaseCurrency(...)`
- `addPricingModule(...)`
- `addAsset(...)`
- `batchProcessWithdraw(...)`
- `batchProcessDeposit(...)`

Moreover, this contract also provides 6 (six) `getters` which are not described here.

4.1.4 `Liquidator`

The `Liquidator` holds the execution logic and storage of all information about liquidating Arcadia Vaults. For example, a Creditor can start an auction to liquidate the collateral of a vault, and any user (the bidder) can buy a vault and end the auction. This contract implements a dutch auction in which the price constantly decreases and the first bidder to buy the vault, immediately ends the auction. The public functions implemented in this contract are listed below:

- `setWeights(...)`
- `setAuctionCurveParameters(...)`
- `setStartPriceMultiplier(...)`
- `setMinimumPriceMultiplier(...)`
- `startAuction(...)`
- `buyVault(...)`
- `endAuction(...)`

In addition to these functions presented above, this contract also provides 3 (three) `getters` which are not described here.

4.1.5 OracleHub

The contract `OracleHub` is responsible for storing the addresses and other necessary information of the Price Oracles and returns asset rates. Only `MainRegistry`, sub-registries, or the contract owner should directly interact with the Oracle Hub. The public functions implemented in this contract are listed below:

- `addOracle(...)`
- `decommissionOracle(...)`

In addition to these functions presented above, this contract also provides 2 (two) `getters` which are not described here.

4.2 Arcadia Lending

Fig.3 presents the essential Arcadia Lending contracts and their main dependencies. In summary, Arcadia Lending is built on top of the Arcadia Vaults. The contracts `Vault`, `Factory`, and `Liquidator` are Arcadia Vaults' contracts that the Lending interacts with. The core contracts in the Lending protocol are `Tranche` and `LendingPool`.

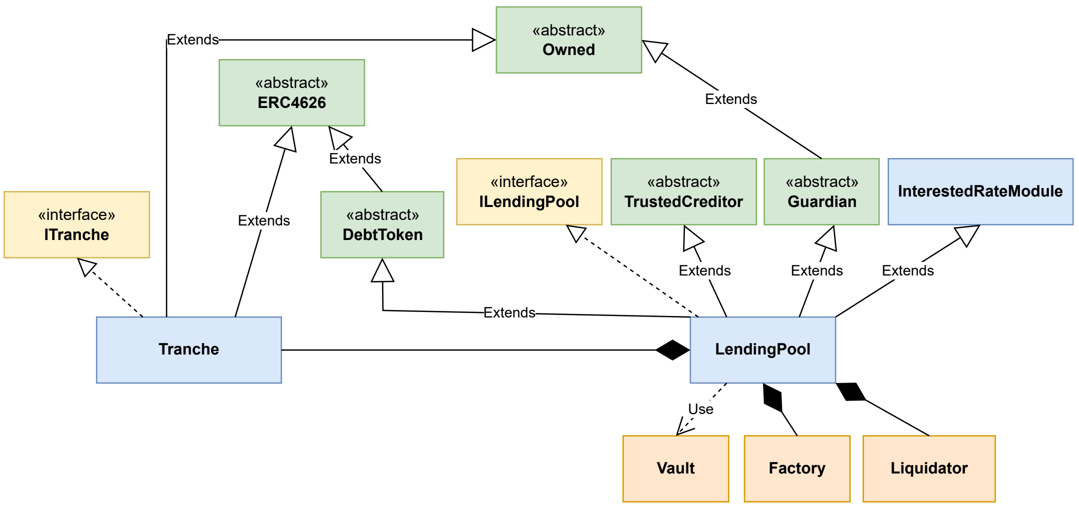


Fig. 3: Arcadia Lending - Structural Diagram of the Contracts

4.2.1 Tranche

The contract `Tranche` extends the `Owned` and `ERC4626` contracts. The ERC-4626 standard is used to represent the interest-bearing positions of liquidity providers. `Tranche` allows for the lending of a specified ERC20 token, managed by a lending pool. This contract provides only the `get` function `totalAssets(...)`. The public functions implemented in this contract are listed below:

- `lock(...)`
- `unLock(...)`
- `setAuctionInProgress(...)`
- `deposit(...)`
- `mint(...)`
- `withdraw(...)`
- `redeem(...)`
- `totalAssetsAndSync(...)`
- `convertToSharesAndSync(...)`
- `convertToAssetsAndSync(...)`
- `previewDepositAndSync(...)`
- `previewMintAndSync(...)`
- `previewWithdrawAndSync(...)`
- `previewRedeemAndSync(...)`

4.2.2 `LendingPool`

The contract `LendingPool` extends the contracts `DebtToken`, `TrustedCreditor`, `Guardian`, and `InterestRateModel`. The ERC-4626 standard is used to represent the open debt of borrowers (`DebtToken`). The abstract contract `TrustedCreditor` implements the minimum functionality to interact with Arcadia Vaults (we omitted this association in Fig.2 for simplicity reasons). The abstract contract `Guardian` provides a mechanism that allows authorized accounts to trigger an emergency stop. Finally, the contract `InterestRateModel` consists of functions to set the configuration parameters and calculate the interest rate.

`LendingPool` implements the main logic to provide liquidity and take or repay loans for a certain asset. This contract does the accounting of the debtTokens (ERC4626) and also provides liquidity against positions backed by Arcadia Vaults as collateral. The public functions implemented in this contract are listed below:

- `addTranche(...)`
- `setInterestWeight(...)`
- `setLiquidationWeight(...)`
- `setMaxInitiatorFee(...)`
- `setTreasuryInterestWeight(...)`
- `setTreasuryLiquidationWeight(...)`
- `setTreasury(...)`
- `setOriginationFee(...)`
- `setBorrowCap(...)`
- `setSupplyCap(...)`
- `depositInLendingPool(...)`
- `donateToTranche(...)`
- `withdrawFromLendingPool(...)`
- `approveBeneficiary(...)`
- `borrow(...)`
- `repay(...)`
- `doActionWithLeverage(...)`
- `liquidityOfAndSync(...)`
- `setInterestConfig(...)`
- `liquidateVault(...)`
- `settleLiquidation(...)`
- `setVaultVersion(...)`

In addition to these functions presented above, this contract also provides 5 (five) `getters` which are not described here.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**. **Likelihood** is a measure of how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding other factors are also considered. These can include but are not limited to: Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if the finding were to be exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Critical] Unused collateral can be stolen from vaults through the lending pool

File(s): `LendingPool.sol` , `Vault.sol`

Description: The lending pool function `doActionWithLeverage(...)` allows users to borrow assets and execute some arbitrary actions through a vault, using the vault's collateral. A condition that must be met after this action is that the vault should be in a healthy state. The `doActionWithLeverage(...)` function can be called by the vault owner, or by an approved beneficiary with some allowance. This allowance check relies on the Solidity 0.8 overflow feature to prevent spending more allowance than a caller has. When the amount is zero, this check will pass as no underflow occurs. The allowance check is shown below:

```
//Check allowances to take debt
if (vaultOwner != msg.sender) {
    uint256 allowed = creditAllowance[vault][msg.sender];
    if (allowed != type(uint256).max) {
        creditAllowance[vault][msg.sender] = allowed - amountBorrowedWithFee;
    }
}
```

Following the allowance check, the vault debt will be updated and the borrowed assets will be transferred, but only if the `amountBorrowedWithFee` is not equal to zero. When this function is called with an `amount` of zero, the `amountBorrowedWithFee` will also be zero, so none of the debt updates or asset transfers will occur. This check is shown below:

```
if (amountBorrowedWithFee != 0) {
    //Mint debt tokens to the vault, debt must be minted Before the actions in the vault are performed.
    _deposit(amountBorrowedWithFee, vault);

    //Send Borrowed funds to the actionHandler
    asset.safeTransfer(actionHandler, amountBorrowed);

    realisedLiquidityOf[treasury] += amountBorrowedWithFee - amountBorrowed;

    emit Borrow(vault, referrer, amountBorrowedWithFee);
}
```

This means that any caller who passes an `amount` of zero will pass all checks and state updates, but the target vault's `vaultManagementAction(...)` function will still execute with the `actionHandler` and `actionData` of the caller's choosing. An attacker could use this behavior to execute arbitrary calls on behalf of any vault with its trusted creditor set to a lending pool, as long as after the arbitrary call the vault is still considered healthy. A possible action an attacker could do is transfer all unused collateral to their own address.

Recommendation(s): Ensure that `amount` cannot be zero in `doActionWithLeverage(...)`.

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-lending/pull/69 . The proposed fix would disallow users to send a transaction that takes 0 additional debt. Although users should in that case directly call the vault, it should remain a valid transaction. The implemented resolution is to enforce that anyone calling the doActionWithLeverage who is not the owner of the vault, has an infinite `creditAllowance` for that vault. We thus assume that a user that sets an 'infinite approval for allowance' for a third party also trusts that third party with the funds in their vault (as the third party could lock the entire vault either way by taking additional debt in the name of the vault).

6.2 [Critical] Vault's `baseCurrency` may differ from the one set by the TrustedCreditor

File(s): `Vault.sol` , `LendingPool.sol`

Description: When a vault opens a new margin account, it sets its base currency to the one selected by the TrustedCreditor. However, until the vault has any used margin, the owner can change the base currency of the vault. The function that allows this change is shown in the code snippet below.

```
function setBaseCurrency(address baseCurrency_) external onlyOwner {
    require(getUsedMargin() == 0, "V_SBC: Non-zero open position");
    _setBaseCurrency(baseCurrency_);
}
```

This difference between currencies could cause multiple issues. Two different scenarios where this issue could affect the protocol are described below.

Scenario 1: A user can borrow more assets than their collateral. When a user borrows assets, a check is done to ensure that the vault used as collateral is in a healthy state after accruing the new debt. To check if a vault is healthy at a specific time, the vault's collateral value is compared with the vault's used margin. This check can be found multiple times throughout the code and can be seen in the code snippet below.

```
...
uint256 usedMargin = getUsedMargin();
if (usedMargin > 0) {
    uint256 collValue = getCollateralValue();
    require(collValue >= usedMargin, "V_VMA: coll. value too low");
}
...
```

The `getCollateralValue()` and `getUsedMargin()` functions can be seen in the snippet of code below.

```
function getCollateralValue() public view returns (uint256 collateralValue) {
    // @audit - This function returns the assets in the vault
    (...) = generateAssetData();

    // @audit - This function will return the collateral value of all the assets in the vault
    collateralValue =
        IMainRegistry(registry).getCollateralValue(...);
}

function getUsedMargin() public view returns (uint256 usedMargin) {
    if (!isTrustedCreditorSet) return 0;

    //getOpenPosition() is a view function, cannot modify state.
    usedMargin = ITrustedCreditor(trustedCreditor).getOpenPosition(address(this));
}
```

As can be seen from the code, the result of the `getCollateralValue()` function will be denominated in the vault's base currency. The result of the function `getUsedMargin()` depends on the TrustedCreditor. In the current implementation, the LendingPool contracts occupy this role. The LendingPool will return the debt accrued to the vault, and it will be denominated in the underlying token of the LendingPool. As mentioned before, these currencies may be different, causing the health check for the vault to return an incorrect value. For example, if the user has `1 ETH` as collateral, it would have a nominal value of `1e18`. If the underlying asset of the LendingPool is USDC, which has 6 decimals, the user could potentially borrow millions of `USDC` with just `1 ETH` as collateral.

Scenario 2: Assets can be locked in the LendingPool. When liquidating a vault, the price of it is paid in its base currency. This payment is then transferred to the LendingPool, where it is added to the realized liquidity. However, if the base currency of the vault is different from the underlying asset of the LiquidityPool, the LiquidityPool cannot transfer or use these assets.

Recommendation(s): Allow to change the `baseCurrency` of the `Vault` only if there is no open margin account

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/235 .

6.3 [Critical] Vaults can borrow from a lending pool different from their trusted creditor

File(s): LendingPool.sol

Description: A vault can interact with a lending pool even if it does not match its trusted creditor. When borrowing assets using `borrow(...)` or `doActionWithLeverage(...)`, debt tokens are minted to the borrowing vault. Vaults are allowed to borrow optimistically, where a health check through the function `isVaultHealthy(...)` is done after the debt has been given. The vault function `isVaultHealthy(...)` only checks the debt tokens of its current trusted creditor. In the case where you borrow from a lending pool different from your trusted creditor, your vault's used margin will not change. This allows a user to borrow unlimited assets from any pool without becoming undercollateralized.

Recommendation(s): When borrowing from a vault, ensure that the vault's trusted creditor is equal to the lending pool address.

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/234 , and https://github.com/arcadia-finance/arcadia-lending/pull/66 .

6.4 [High] Privileged vault roles not reset upon ownership change

File(s): `Vault.sol`

Description: A vault has two privileged roles: Beneficiary and Asset Manager. The beneficiary role can borrow on behalf of the vault, and the asset manager role can execute arbitrary calls using the vault's funds. Upon ownership change such as through liquidations or selling vaults, the vault `owner` is updated, however, the privileged roles are not reset. Since previous privileged roles were not cleared, the new owner may have their assets transferred or borrowed by addresses trusted by the previous owner, but not the new owner. This behavior can be combined with a liquidation event to steal more funds than were originally in the vault. When a liquidation event starts, the debt tokens belonging to the liquidated vault are set to zero as shown below in `LendingPool`:

```
function liquidateVault(address vault) external whenLiquidationNotPaused processInterests {
    ...
    //Remove debt from Vault (burn DebtTokens)
    _withdraw(openDebt, vault, vault);
}
```

During liquidation, the owner is also removed from the vault. Since the collateral is still in the vault but the debt has been set to zero, the privileged roles can use the assets again. This allows for two scenarios depending on the role:

- **For the beneficiary role,** when liquidation starts it is possible to call `borrow(...)` and `doActionWithLeverage(...)` while the auction is still in progress. These funds can be borrowed without the intention to return, allowing the original owner to keep their borrowed funds, and then borrow using their collateral once again;
- **For the asset manager role**, when liquidation starts it is possible to call `vaultManagementAction(...)` to execute an arbitrary call to transfer the collateral tokens. Since the debt has been reset, the under-collateralization check will pass and the tokens will be transferred. This allows the original owner to keep their borrowed funds, and then recover their collateral as well.

Recommendation(s): Ensure that the privileged roles are cleared during liquidation or when ownership is transferred.

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/237 , and https://github.com/arcadia-finance/arcadia-lending/pull/67 .

6.5 [Medium] Incorrect calculation of `openDebt` during Vault liquidation

File(s): `LendingPool.sol`

Description: To initiate a liquidation for a vault, the function `liquidateVault(...)` in the LendingPool contract must be called. This function performs the following actions:

- Calculate the `openDebt` for the vault;
- Initiate an auction for the vault;
- Remove the debt accrued by the vault;

The `openDebt` value is a crucial component of the liquidation process. It is used to calculate the liquidation price and to determine if bad debt exists after liquidation. The following line of code is used to calculate the `openDebt` for a vault.

```
uint256 openDebt = balanceOf[vault];
```

However, the balance in the LendingPool refers to the DebtTokens owned by the vault, not the outstanding debt for that vault. Using DebtTokens instead of the actual debt would cause the `openDebt` used for the liquidation process to be smaller than the actual debt accrued by the vault. This has multiple consequences, such as:

- Vaults may not be liquidated even if they have already reached their liquidation threshold;
- Because not all the debt of the vault will be liquidated, the vault will keep some debt even after liquidation ;

Recommendation(s): Instead of setting `openDebt` equals to `balanceOf[vault]`, consider setting it to `maxWithdraw(vault)`, as shown below.

```
- uint256 openDebt = balanceOf[vault];
+ uint256 openDebt = maxWithdraw(vault);
```

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-lending/pull/68 What essentially would happen here is a partial liquidation: only part of the `openDebt` is withdrawn (the realized portion), and only for this part of the `openDebt`, the auction price is calculated. In practice, this would be only a very limited difference, being the accrued interests. Although this was unintended behavior, this wouldn't lead to user or protocol loss of funds. The interests that have been added to the realized liquidity would remain as open debt on the vault.

Imagine the scenario:

- A user opens a vault, and deposits assets worth 11. ;
- The user takes out a debt of 10. ;
- One year progresses at approximately 100% interest per year. ;
- The vault owner however has kept depositing assets in his vault to prevent being liquidated;
- Situation: balanceOf = 10, actual debt = 11 ;

Now the vault gets liquidated, and it gets auctioned for 10. Now, the vault buyer receives a vault, but it still has 1 debt. There is no loss of funds and no errors in accounting, although we do agree that this was an unforeseen scenario.

6.6 [Medium] Pricing module exposure can be manipulated for WETH token

File(s): Vault.sol

Description: When depositing or withdrawing tokens to a vault, the user passes an array assetTypes as an argument to specify which type of asset its associated address in the assetAddresses array is. The asset type will determine which internal deposit or withdrawal function to use. However, there are no checks to ensure that an address matches the type passed by the user. During a deposit, it is possible for a user to pass an ERC20 token address, but set the asset type to indicate that it's an ERC721 token. This would lead to _depositERC721(...) being called, which would attempt to call safeTransferFrom(...) on the ERC20 token which does not exist. For most tokens, this would revert and no damage to the protocol can be done. The function is shown below:

```
function _depositERC721(address from, address ERC721Address, uint256 id) internal {
    IERC721(ERC721Address).safeTransferFrom(from, address(this), id);

    erc721Stored.push(ERC721Address);
    erc721TokenIds.push(id);
}
```

For the token WETH a different behavior will occur. WETH implements a fallback function that takes the msg.value and increments the caller's balance. In the case of the safeTransferFrom call above, this would be a zero value call so no balance change would occur, but the call would still succeed rather than reverting. The fallback function is shown below:

```
function() public payable {
    deposit();
}

function deposit() public payable {
    balanceOf[msg.sender] += msg.value;
    Deposit(msg.sender, msg.value);
}
```

An attacker can pass WETH for a deposit or withdrawal stating that it is an ERC721 token, and use the fallback behavior of WETH to successfully call deposit(...) and withdraw(...) without transferring any balance. In both functions, the following call to update the pricing module exposure will still occur, as shown below:

```
// When depositing
IMainRegistry(registry).batchProcessDeposit(assetAddresses, assetIds, assetAmounts);

// When withdrawing
IMainRegistry(registry).batchProcessWithdrawal(assetAddresses, assetIds, assetAmounts);
```

This allows an attacker to arbitrarily manipulate the exposure of the WETH pricing module. The exposure can be set to zero, in which case any further withdrawals by other users will lead to reverting as they attempt to subtract their withdrawal amount from zero. This would lead to users' funds being permanently locked. The exposure can also be increased to the max exposure, preventing any further deposits from other users and acting as a deposit DOS. The exposure can also be set to any value in between, allowing for subtle changes in exposure that result in higher or lower levels of risk than expected.

Recommendation(s): Check that asset types are correct for each asset.

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/240 . The team has assessed the High classification and deems this to be too high. Although this could temporarily block users' funds, it can be relatively easily mitigated by bundling a flash bots transaction where the exposure is increased together with a withdrawal call. Alternatively, users could upgrade to a new vault version where the batch process is skipped and where debt can only be repaid, and as such allow users to withdraw assets. This would not lead to any user (borrower or LP) nor protocol losses.

Update from Nethermind: After the discussion with the client, we agreed to downgrade the severity to Medium.

6.7 [Medium] Vault deployment susceptible to Denial of Service Attacks

File(s): Factory.sol

Description: Users deploy their vaults through the function createVault(...) which uses the CREATE2 opcode to deploy the vault to a deterministic address. The salt used for deployment is controlled by the caller through the argument salt. It is possible for a user's vault deployment transaction to be front-run by an attacker, where the attacker's transaction uses the same salt argument. This would lead to the attacker's transaction executing first, making them the owner of the new contract. When the user's transaction executes the vault deployment will fail as the attacker's vault is already deployed at that address. This can be used for DOS vault deployment against any user or allow a targeted DOS against a specific address (for example, a particular ENS domain address).

Recommendation(s): Consider mixing the user-provided salt argument with msg.sender to ensure that deployed vault addresses will still be deterministic per caller, but two different callers with the same salt will not get the same vault address.

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/242 . Note that we used `tx.origin` instead of `msg.sender`, as such even vaults deployed via a third contract cannot be front-runned.

6.8 [Medium] `minPriceMultiplier` is not scaled to 20 decimals in `_calcPriceOfVault(...)`

File(s): `Liquidator.sol`

Description: The `minPriceMultiplier` is a two-decimal value used to calculate the vault pricing during an auction. For example, a minimum price multiplier of 30 would represent 30%. This value is used by `_calcPriceOfVault(...)` to determine the pricing of a vault based on the current time and cut-off time. When the cut-off time has not been reached, the following formula is used:

```
price = openDebt * (LogExpMath.pow(base, timePassed) * (startPriceMultiplier - minPriceMultiplier) +
↪ minPriceMultiplier) / 1e20;

// NOTE: timepassed = (block.timestamp - startTime) * 1e18
```

The final `minPriceMultiplier` which is only two decimals is added to an 18-decimal value. When the division by 1e20 occurs, the added `minPriceMultiplier` is insignificant and the integer division will disregard the value completely. This causes the sum `minPriceMultiplier` to effectively play no part in the final price calculation. This impacts the vault auction price by causing it to be lower than expected, which is demonstrated below.

Scenario: Consider the following values set in the Liquidator contract:

```
startPriceMultiplier = 110      // 2 decimals
minPriceMultiplier = 50         // 2 decimals
cutoffTime = 14400
base = 1e18                     // 18 decimals
```

Let's assume that 6000 seconds have passed, so approximately 42% of the time before `cutoffTime`:

```
timepassed = 6000 * 1e18 = 6000000000000000000000       // 18 decimals
openDebt = 10000
```

We would expect the resulting price to simply be `openDebt * startPriceMultiplier` because the base is `1e18` and the `cutoffTime` has not been exceeded. However, we actually calculate the price to be:

```
price = openDebt * (LogExpMath.pow(base, timePassed) * (startPriceMultiplier - minPriceMultiplier) +
↪ minPriceMultiplier) / 1e20;
price = 10000 * (LogExpMath.pow(1e18, 6000000000000000000000) * (110 - 50) + 50) / 1e20;
price = 10000 * (1e18 * (60) + 50) / 1e20;
price = 10000 * (6e19 + 50) / 1e20;
price = (6e23 + 500000) / 1e20;
price = 6000;
```

Since the `base` is 1e18 in our scenario, we do not expect the price to decrease regardless of the time elapsed. The calculated price should always be `openDebt * startPriceMultiplier = 11000` in our example. However, the price that we calculated was 6000, which is incorrect. If we run the same calculation but adjust the sum `minPriceMultiplier` to be 20 decimals we will receive the proper vault price:

```
price = openDebt * (LogExpMath.pow(base, timePassed) * (startPriceMultiplier - minPriceMultiplier) +
↪ (minPriceMultiplier * 1e18)) / 1e20;
price = 10000 * (LogExpMath.pow(1e18, 6000000000000000000000) * (110 - 50) + 5e19) / 1e20;
price = 10000 * (1e18 * (60) + 5e19) / 1e20;
price = 10000 * (6e19 + 5e19) / 1e20;
price = (6e23 + 5e23) / 1e20;
price = 11e23 / 1e20;
price = 11000;
```

We can observe that when `minPriceMultiplier` is not brought up to 20 decimals, the purchase price of the vault in an auction is lower than expected.

Recommendation(s): Ensure that the sum for `minPriceMultiplier` is scaled to 18 decimals.

```
- price = openDebt
-     * (LogExpMath.pow(base, timePassed) * (startPriceMultiplier - minPriceMultiplier) + minPriceMultiplier)
-     / 1e20;
+ price = openDebt
+     * (LogExpMath.pow(base, timePassed) * (startPriceMultiplier - minPriceMultiplier) +
↪ uint256(minPriceMultiplier)*1e18)
+     / 1e20;
```

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/238 . The team has assessed the `High` classification and argues that this classification is too high. A first -and valid- implementation of Arcadia's auction curves was to let auction prices go to zero altogether.

Although the `minPriceMultiplier`'s decimals are not correct in this implementation, the auction price curve would still pass all price levels and would thus still be able to be sold to atomic liquidators at the market- or similar price. Besides that, we possess multiple other degrees of freedom by using other auction curve parameters to create a similar curve as the curve that was expected when `minPriceMultiplier` had the foreseen decimals. This issue is relatively easily mitigated and we do not see this issue causing protocol-wide (or any) loss. We thus propose a `Low` or `Medium` classification.

Update from Nethermind: After the discussion with the client, we agreed to downgrade the severity to `Medium`.

6.9 [Low] Asset can be added with an inactive oracle or with no oracles

File(s): `OracleHub.sol`

Description: New assets are added to the main registry by pricing modules with the function `addAsset(...)`. Assets' additional data are also stored in mapping `assetToInformation` in the pricing module. Before that, the array `oracles` is checked with a call to `OraclesHub.checkOracleSequence(`
However, the function does not check if the array `oracles` is not empty, which allows it to add assets without oracles. The oracles are also not checked if they are active, so assets may be added with inactive oracles.

New assets are added to the main registry through a pricing module contract with the function `addAsset(...)`. When adding an asset the caller must provide a sequence of oracles, which are validated in `OracleHub.checkOracleSequence(...)`. However, there are no checks to ensure the following conditions: **a) the provided array of oracles must not be empty; b) all of the provided oracles must be active.**

Recommendation(s): Check-in `OraclesHub.checkOracleSequence(...)` if the array `oracles` is not empty, and if the provided oracles are active.

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/257 .

6.10 [Low] Asset can be added with an incorrect oracle sequence

File(s): `StandardERC20PricingModule.sol`

Description: The function `addAsset(...)` will add a new `asset` to the `MainRegistry`. As part of the `asset` information, this function receives a sequence of oracles to price the `asset` in `USD`. This sequence of oracles is checked in the function `checkOracleSequence(...)` from the `OracleHub` contract. The check is shown in the code snippet below.

```
function addAsset(...)
    external
    onlyOwner
{
    require(!inPricingModule[asset], "PM20_AA: already added");

    IOraclesHub(oracleHub).checkOracleSequence(oracles);

    inPricingModule[asset] = true;
    assetsInPricingModule.push(asset);

    uint256 assetUnit = 10 ** IERC20(asset).decimals();
    require(assetUnit <= 1e18, "PM20_AA: Maximal 18 decimals");

    assetToInformation[asset].assetUnit = uint64(assetUnit);
    assetToInformation[asset].oracles = oracles;
    _setRiskVariablesForAsset(asset, riskVars);

    exposure[asset].maxExposure = maxExposure;

    IMainRegistry(mainRegistry).addAsset(asset);
}
```

The `checkOracleSequence(...)` function ensures that the provided oracle sequence meets the following requirements:

- The sequence must not contain more than three oracles;
- Each oracle address, must have been added previously to the `OracleHub`;
- The `baseAsset` of each oracle must be equal to the `quoteAsset` of the next oracle;

However, there is no check ensuring that the `quoteAsset` of the first oracle is the same as the asset being added. If the first oracle of the sequence has an incorrect `quoteAsset`, the returned price could be wrong.

Recommendation(s): Consider adding a check to ensure that the `quoteAsset` from the first oracle is the same as the asset being added, otherwise revert.

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/257 .

6.11 [Low] Liquidator contract owner can manipulate live auction pricing

File(s): `Liquidator.sol`

Description: The owner of the `Liquidator` contract has access to the function `setMinimumPriceMultiplier(...)` which allows the `minPriceMultiplier` to be set to a value within the range of 0 to 90. When a vault is on auction, the price is found using `_calcPriceOfVault(...)` which uses `minPriceMultiplier` as part of the price calculation.

If the `Liquidator` owner address becomes compromised (private key breach or multi-sig issue), since `setMinimumPliceMultiplier(...)` can be called at any time by the owner, it is possible for the owner to abuse the `minPriceMultiplier` by setting it to a very low value, purchasing the vault, and then returning the value back to normal.

As described below in `_calcPriceOfVault(...)`, while `timePassed` is not greater than `cutoffTime`, the vault price continuously decreases. If `minPriceMultiplier` is set during an active auction, the prices can decrease faster than expected.

```
function _calcPriceOfVault(uint256 startTime, uint256 openDebt) internal view returns (uint256 price) {
    uint256 timePassed;

    unchecked {
        timePassed = block.timestamp - startTime;

        if (timePassed > cutoffTime) {
            price = openDebt * minPriceMultiplier / 1e2;
        } else {
            timePassed = timePassed * 1e18;

            price = openDebt
                * (LogExpMath.pow(base, timePassed) * (startPriceMultiplier - minPriceMultiplier) + minPriceMultiplier)
                / 1e20;
        }
    }
}
```

Recommendation(s): Consider setting some lower limit to the `minPriceMultiplier` in `setMinimumPriceMultiplier(...)` or prevent `minPriceMultiplier` from being changed while an auction is live.

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/259 . The implemented solution stores all auction curve data per auction, and as such the owner has no influence on the price curve of any ongoing auction. We do like to mention that this finding implies the private keys of n of the n-out-of-m multisign signers of the liquidator contract are compromised.

6.12 [Low] Oracle's data is not checked to be up-to-date

File(s): `MainRegistry.sol` , `OracleHub`

Description: The function `getListOfValuesPerAsset(...)` may use ChainLink oracles to compute some of the asset's values. However, the values returned by these oracles are not checked to be up to date. Relevant parts of the function are reproduced in the snippet of code below.

```
function getListOfValuesPerAsset(...) public view returns (RiskModule.AssetValueAndRiskVariables[] memory) {
    ...
        if (rateBaseCurrencyToUsd == 0) {
        //Get the BaseCurrency-USD rate
        // @audit - This does not ensure that the data is not stale
        (, rateBaseCurrencyToUsd,,,) = IChainLinkData(
            baseCurrencyToInformation[baseCurrency].baseCurrencyToUsdOracle
        ).latestRoundData();
    ...

```

Using a stale value could affect the returned values.

Recommendation(s): Consider checking if the fetched data was updated recently enough.

Status: Mitigated.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/244 . I used a similar pattern to also allow the owner of the ERC20PricingModule to update price oracles for an asset, but only if one of the oracles was decommissioned first: https://github.com/arcadia-finance/arcadia-vaults/pull/258 .

6.13 [Low] Rounding issues in DebtToken shares calculation can steal funds from LendingPool

File(s): `DebtToken.sol`

Description: When users borrow funds from the LendingPool, an amount of DebtToken is minted for the user based on the current rate between the underlying asset and share. However, when calculating the number of shares, the current implementation rounds up for deposits and rounds down for withdrawals. This rounding issue can be exploited by attackers to steal funds from the LendingPool. For example, let's say that Alice is the first user to borrow from the LendingPool, and she borrows `amount = 1000`. The state of the pool would be:

```
realisedDebt = 1000
balanceOf[Alice] = 1000
totalSupply = 1000
```

Sometime after that, Bob borrows `amount = 95` from the LendingPool. Because some time had passed, there is an unrealized debt of 100, so the current state of the pool is:

```
realisedDebt = 1000
unrealisedDebt = 100
=> totalAssets() = 1100
```

The amount of DebtToken received by Bob is computed in the following way:

```
shares = mulDivDown(assets, totalSupply, totalAssets)
       = roundDown((95 * 1000) / 1100)
       = roundDown(86.3) = 86
```

After borrowing, the LendingPool is in the following state:

```
totalAsset() = realisedDebt + unrealisedDebt + 86 = 1186
balanceOf[Alice] = 1000, balanceOf[Bob] = 86
totalSupply = 1086
```

If in the same block, Bob tries to repay `amount = 93`, the amount of DebtTokens that will be burned is computed in the following way:

```
shares = mulDivUp(assets, totalSupply, totalAsset)
           = roundUp((93 * 1086) / 1186)
           = roundUp(85.15)
           = 86
balanceOf[Bob] = 0
```

So, he will have repaid less than he borrowed, even though all his Debt Tokens were burnt.

Recommendation(s): To prevent this issue, we recommend rounding up in function `_deposit(...)` and rounding down in function `_withdraw(...)` when calculating shares.

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-lending/pull/70 , and https://github.com/arcadia-finance/arcadia-vaults/pull/245 .

6.14 [Low] The second-most junior tranche can avoid loss by front-running the function `settleLiquidation(...)`

File(s): `LendingPool.sol`

Description: Arcadia Lending has been designed such that liquidity providers are able to choose distinct risk tranches that align with their respective risk profiles. In the event of any defaults, the most junior tranche will experience a reduction in their underlying tokens. Additionally, if the most junior tranche is entirely wiped out, the next most junior tranche will assume responsibility, and so on. However, the current implementation only locks the most junior tranche during liquidation auctions. This could potentially allow liquidity providers from other tranches to withdraw their funds as soon as the auction price went down and had potential effects on their own tranche.

Recommendation(s): Consider locking all tranches during the liquidation auction.

Status: Acknowledged.

Update from the client: This is a design choice. Default events should already occur very exceptionally, a complete wipe-out of the junior tranche is an unhappy flow of an unhappy flow and should never occur in practice. We do not want to block the LPs in more senior tranches to avoid certain from frontrunning in a scenario that in theory should never occur.

6.15 [Low] Vault asset limit can be bypassed with `vaultManagementAction(...)`

File(s): `Vault.sol`

Description: Each vault has an asset limit to ensure that users cannot deposit too many assets. This check exists in the function `deposit(...)` and is shown below:

```
function deposit(...) external onlyOwner {
    uint256 assetAddressesLength = assetAddresses.length;

    require(
        assetAddressesLength == assetIds.length && assetAddressesLength == assetAmounts.length
        && assetAddressesLength == assetTypes.length,
        "V_D: Length mismatch"
    );

    _deposit(assetAddresses, assetIds, assetAmounts, assetTypes, msg.sender);

    require(erc20Stored.length + erc721Stored.length + erc1155Stored.length <= ASSET_LIMIT, "V_D: Too many assets");
}
```

However, this asset limit can be bypassed through the function `vaultManagementAction(...)`, as it makes a call directly to `_deposit(...)` without checking if the asset limit has been reached.

Recommendation(s): Ensure that the asset limit is checked both during calls to `deposit(...)` and `vaultManagementAction(...)`.

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/239 .

6.16 [Low] Vault beneficiaries indirectly obtain the asset manager role

File(s): `LendingPool.sol` , `Vault.sol`

Description: The role of asset manager for a vault should only be assigned to fully trusted accounts. Accounts with this role can use the function `vaultManagementAction(...)` to execute arbitrary calls using funds from the vault. With the condition that after executing these calls the vault stays in a healthy state. However, the function `vaultManagementAction(...)` can be also called through the function `doActionWithLeverage(...)` (located in the LendingPool contract) if the LendingPool is the current TrustedCreditor of the vault.

A user should be able to call `doActionWithLeverage(...)` for a vault if it is the owner of the vault, or if it has any credit given for this vault. Credit or allowance for a vault is a quantitative measure of trust, but because any amount of allowance permits the user to execute the `doActionWithLeverage(...)` function, giving the minimum amount of allowance to a user implies full trust over that user.

Recommendation(s): Only allow the owner and asset managers of a vault to execute `doActionWithLeverage(...)` for the vault.

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/237 , https://github.com/arcadia-finance/arcadia-lending/pull/67 , and https://github.com/arcadia-finance/arcadia-lending/pull/69 .

6.17 [Undetermined] ERC721 pricing modules exposure manipulation

File(s): `Vault.sol`

Description: When withdrawing `ERC721` assets from a vault, the internal function `_withdrawERC721(...)` will handle the token transfer, as well as update `erc721Stored` and `erc721TokenIds` if necessary.

It is possible to transfer an ERC721 token directly to the vault contract without using the proper `deposit(...)` function. By skipping `deposit(...)`, the following is not updated:

- The ERC721 pricing module exposure is not increased for the given token collection;
- The token ID is not tracked in the vault's `erc721TokenIds` array;
- The token address is not tracked in the vault's `erc721Stored` array;

Once a direct ERC721 transfer is done, it is possible to withdraw the token through the proper `withdraw(...)` function, which decreases the token's pricing module exposure even though it was never increased because `deposit(...)` was not called. There are two approaches to successfully decreasing an ERC721 pricing module exposure:

The vault has no tracked ERC721 tokens: An ERC721 token is directly transferred to the vault contract. During the withdrawal process, the `tokenIdLength` will be zero so the logic in the `else` statement will execute. Since the `tokenIdLength` is zero the for loop will execute zero times and the token will be transferred from the vault to the `to` address.

The vault has one tracked ERC721 token: The vault has only one ERC721 token that has been correctly deposited. An ERC721 token of a different collection is then transferred directly to the vault. During the withdrawal process for the directly transferred token, the `tokenIdLength` is one, so `erc721TokenIds` and `erc721Stored` (containing only one entry) are both popped. The directly transferred token is then transferred to the `to` address. Note that now the properly added ERC721 token is not tracked, so approach (1) can be used to recover these tokens (which will reduce the exposure back to the correct level) or the token can be skimmed which will leave the exposure increased.

Both approaches allow the exposure of an ERC721 pricing module to be reduced, which will eventually prevent users from withdrawing due to overflow checks when the exposure tries to decrease from zero. Approach (2) allows for the exposure to be increased, which could be used to reach the max deposit and DOS any deposits by other users. This finding has a severity of undetermined as part of the exploit path involves the ERC721 pricing module which is out of scope, however, this finding is still present in the report as the cause of this issue exists in the in-scope `Vault` contract. The function is shown below:

```
function _withdrawERC721(address to, address ERC721Address, uint256 id) internal {
    uint256 tokenIdLength = erc721TokenIds.length;

    if (tokenIdLength == 1) {
        // there was only one ERC721 stored on the contract, safe to remove both lists
        erc721TokenIds.pop();
        erc721Stored.pop();
    } else {
        for (uint256 i; i < tokenIdLength;) {
            if (erc721TokenIds[i] == id && erc721Stored[i] == ERC721Address) {
                erc721TokenIds[i] = erc721TokenIds[tokenIdLength - 1];
                erc721TokenIds.pop();
                erc721Stored[i] = erc721Stored[tokenIdLength - 1];
                erc721Stored.pop();
                break;
            }
            unchecked {
                ++i;
            }
        }
    }

    IERC721(ERC721Address).safeTransferFrom(address(this), to, id);
}
```

Recommendation(s): Ensure that in `_withdrawERC721` the token being withdrawn and its ID exists in the `erc721Stored` and `erc721TokenIds` arrays.

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/241 .

6.18 [Info] Blocking the latest vault version does not change `latestVaultVersion`

File(s): `Factory.sol`

Description: In the vault factory it is possible for the owner to block a particular vault version from being deployed or upgraded. However in the case that the vault version being blocked is the latest version, the state variable `latestVaultVersion` does not change. This affects the `createVault(...)` function, as an expected feature is that you can pass `0` as the `vaultVersion` to use the latest vault version. When a user does this, a `require` statement checking for a blocked version would revert when a user tries to deploy with the latest vault version, which may confuse users.

It is challenging to address this issue as simply rolling back the latest vault version will lead to the version information getting overwritten on the next vault update. If this approach is taken, the vault upgrade Merkle tree must also support the blocked latest version (for users that deployed the vault before it was blocked).

Recommendation(s): This issue can be addressed by changing the contracts, however, an alternative option is to be prepared to modify the website UI to designate a particular unblocked vault version as the default instead of using the latest.

Status: Mitigated.

Update from the client: This issue will need to be addressed at the level of the website UI, not the smart contracts

6.19 [Info] Closing margin account doesn't reset `liquidator`

File(s): `Vault.sol`

Description: While closing a margin account using `closeTrustedMarginAccount(...)` the values such as `isTrustedCreditorSet` and `trustedCreditor` are reset, while the `liquidator` value is not.

```
function closeTrustedMarginAccount() external onlyOwner {
    require(isTrustedCreditorSet, "V_CTMA: NOT SET");
    //getOpenPosition() is a view function, cannot modify state.
    require(ITrustedCreditor(trustedCreditor).getOpenPosition(address(this)) == 0, "V_CTMA: NON-ZERO OPEN POSITION");

    isTrustedCreditorSet = false;
    trustedCreditor = address(0);
}
```

Recommendation(s): Consider resetting `liquidator` while closing a margin account.

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/254 .

6.20 [Info] Function `changeGuardian(...)` does not emit the `oldGuardian`

File(s): `BaseGuardian.sol` , `Guardian.sol`

Description: The function `changeGuardian(...)` is used to set the guardian address. The function should emit an event with the `oldGuardian` and `newGuardian` addresses. However, the event is triggered after updating the new address, thus, overriding the current guardian.

```
function changeGuardian(address guardian_) external onlyOwner {
    guardian = guardian_;
    ///////////////////////////////////////
    // @audit Event is not emitting the oldGuardian
    //////////////////////////////////
    emit GuardianChanged(guardian, guardian_);
}
```

Recommendation(s): Consider emitting the event `GuardianChanged` before the new guardian address is set or create a temporary variable to save the `oldGuardian` address.

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/250 , and https://github.com/arcadia-finance/arcadia-lending/pull/71 .

6.21 [Info] Functions will default to USD if an invalid `baseCurrency` is used

File(s): `MainRegistry.sol`

Description: The following functions receive an argument `baseCurrency` which indicates which base currency the asset value should be denominated in the functions below.

```
- getListOfValuesPerAssets(address[],uint256[],uint256[],address)
- getCollateralValue(...)
- getLiquidationValue(...)
- getTotalValue(address[],uint256[],uint256[],address)
```

If an invalid `baseCurrency` is used, the mapping `assetToBaseCurrency` will return the value `0`, which is a valid identifier for the base currency `USD`. Users would expect that passing some asset as `baseCurrency` would return the value of the assets denoted in their chosen base currency, however, if an invalid `baseCurrency` is provided, these functions will silently default to USD without giving any warning to the user.

Recommendation(s): Consider adding a check to ensure that the received `baseCurrency` is valid and revert otherwise.

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/248 .

6.22 [Info] Handling of ERC721 Assets May Be Incorrect in Action Handler

File(s): `MultiCall.sol`

Description: The function `ActionMultiCall.executeAction(...)` allows the vault to make arbitrary calls while using its assets. At the end of the function, the field `incoming.assetAmounts` is updated for `ERC20` and `ERC1155` tokens, but not for `ERC721` tokens. While `ERC721` token transfers do not require an amount, the vault still needs the amount for this token to be set to 1. Failing to set the `incoming.assetAmounts` for the `ERC721` token in the `actionData` call may result in the token being left in the action handler contract.

Recommendation(s): Consider setting the amount for `ERC721` tokens to 1 in the `ActionMultiCall.executeAction(...)` function.

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/255 .

6.23 [Info] Initial curve for pricing vaults is not decreasing

File(s): `Liquidator.sol`

Description: In the Liquidator contract constructor, the state variable base is set to `1e18`. This variable is used to compute the vault price in the `_calcPriceOfVault(...)` function. The base determines how quickly the price decreases over time. The vault price is expected to decrease over time in a dutch auction fashion. Therefore, the value returned by the `LogExpMath.pow(base, timePassed)` function should be strictly less than `1`. However, with the current base value, the `LogExpMath.pow(base, timePassed)` function will always return 1, and the vault price will never decrease as long as `timePassed` is less than or equal to `cutoffTime`.

```
constructor(...){
    ...
    base = 1e18;
}

function _calcPriceOfVault(uint256 startTime, uint256 openDebt) internal view returns (uint256 price) {
    ...
    price = openDebt
            * (LogExpMath.pow(base, timePassed) * (startPriceMultiplier - minPriceMultiplier) + minPriceMultiplier)
            / 1e20;
}

```

Recommendation(s): It is recommended to set a decreasing curve for the price of vaults in the constructor.

Status: Acknowledged.

Update from the client: It will be set in the constructor before contracts are deployed.

6.24 [Info] Some common ERC20 tokens can't be deposited

File(s): `Vault.sol`

Description: The function `_depositERC20(...)` checks the return value from token transfer.

```
require(IERC20(ERC20Address).transferFrom(from, address(this), amount), "V_D20: Transfer from failed");
```

That makes a problem for depositing tokens that do not return a success value (e.g. USDT), and the `require` statement would fail even if the transfer was successful. Note that the same issue is in functions `skim(...)` and `_withdrawERC20(...)`.

Recommendation(s): Consider using the `SafeERC20` library to allow for tokens that do not return on a successful transfer. Alternatively, ensure that the whitelisted ERC20 tokens that are allowed to be used as collateral all return `true` on a successful transfer.

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/256 .

6.25 [Info] Unnecessary function `onERC721Received(...)` in factory contract

File(s): `Factory.sol`

Description: The factory contract implements the function `onERC721Receieved(...)` which allows contracts to receive ERC721 tokens. Since the factory is an ERC721 token itself, it should not need to hold any token, and transferring any ERC721 token to the factory will result in a loss of funds.

Recommendation(s): Consider removing the function `onERC721Received(...)` in the factory contract.

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/247 .

6.26 [Info] `assetIds` length may differ from `amounts`

File(s): `MainRegistry.sol`

Description: The function `batchProcessWithdrawal(...)` receives arrays as parameters that need to have the same length. However, there is no check for the array `assetIds` length.

```solidity
function batchProcessWithdrawal(
    address[] calldata assetAddresses,
    uint256[] calldata assetIds,
    uint256[] calldata amounts
) external whenWithdrawNotPaused onlyVault {
    uint256 addressesLength = assetAddresses.length;
    require(addressesLength == amounts.length, "MR_BPW: LENGTH_MISMATCH");
    ...
}
```

Recommendation(s): Consider comparing the length of the array `assetIds` with `assetAddresses` or `amounts` arrays.

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/251 . This would in practice never have been a problem since there were redundant checks on the length of arrays in the `deposit()` and `withdraw()` in `Vault.sol`. We have however also removed these redundant checks in commit hash `ff07b19d2618f9606b17d1e69da4703ae0bd182c` .

6.27 [Best Practices] Possible optimization in `getListOfValuesPerAsset(...)`

File(s): `MainRegistry.sol`

Description: In the function `getListOfValuesPerAsset(...)` the rate between the `baseCurrency` and USD is received from the ChainLink oracle. This is only fetched once during the function call, but every time a comparison is made to check that the value has been fetched, to avoid fetching again.

```solidity
function getListOfValuesPerAsset(
    address[] calldata assetAddresses,
    uint256[] calldata assetIds,
    uint256[] calldata assetAmounts,
    uint256 baseCurrency
) public view returns (RiskModule.AssetValueAndRiskVariables[] memory) {
    ...
    if (rateBaseCurrencyToUsd == 0) {
        //Get the BaseCurrency-USD rate
        (, rateBaseCurrencyToUsd,,,) = IChainLinkData(
            baseCurrencyToInformation[baseCurrency].baseCurrencyToUsdOracle
        ).latestRoundData();
    }
    ...
```

Recommendation(s): Consider fetching this value outside the loop and save it in a memory variable for use in the loop.

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/253 . While doing the optimization I noticed another possible issue when both `valueInUsd` and `valueInBaseCurrency` are non-zero, see commit hash `92f0fc1003c0fa739ef4d374b00c20af60132a0f` , which is also part of previous mentioned PR.

6.28 [Best Practices] Storage load optimizations

File(s): `Vault.sol`

Description: **Description**: Two potential optimizations in the vault contract have been identified that involve caching storage variables to save gas. In the vault contract, the length of an array can be cached outside a loop to save gas, as shown below.

```diff
- for (uint256 i; i < erc721Stored.length;)
+ uint256 erc721StoredLength = erc721Stored.length;
+ for (uint256 i; i < erc721StoredLength;)
```

In the same contract, a temporary variable is used to reduce storage reads in the `generateAssetData(...)` function. Consider doing the same for the token id of `ERC1155`.

Recommendation(s): Consider caching these storage variables to reduce storage reads.

Status: Fixed.

Update from the client: Fixed in https://github.com/arcadia-finance/arcadia-vaults/pull/252 .

7 Documentation Evaluation

Technical documentation is created to explain what the software product does. This way, developers and stakeholders can easily follow the purpose and the underlying functionality of each file/function/line. Documentation can come not only in the form of a README.md but also using code as documentation (to write clear code), diagrams, websites, research papers, videos, and external documentation. Besides being a good programming practice, proper technical documentation improves the efficiency of audits. Less time can be spent understanding the protocol and more time can be put towards auditing which improves the efficiency and overall output of the audit.

The **Arcadia** team provided three documents to assist the audit process: **(a) Arcadia Finance -** specifies the problem, the solution, and the product; **(b) Arcadia Vaults -** includes information on its architecture, pricing logic, asset management, and vault logic; and **(c) Arcadia Lending -** embraces a diagram describing the flow of the lending application, risk models, features that are implemented (e.g., liquidations, risk tranches, interests, etc.), and how they should behave for different scenarios.

These documents covered the most common terms used in the source code, explanations for architecture decisions, and core business logic and functions flow. By reading the whole documentation suite, we could get a proper comprehension of how the contracts should operate. The provided documentation is well-written and structured. It is a complete source of resources for developers and auditors. The codebase has sufficient inline comments and meaningful naming for functions and variables, helping the audit team to understand function flow and to detect issues.

7.1 Documentation inconsistencies

Along this investigation, we noticed some areas in the code where the implementation differs from the specification and/or inline comments.

7.1.1 OracleHub documentation is contradictory

The `OracleHub` contract documentation stipulates that no end-user should interact with the contract. However, `checkOracleSequence` and `decommissionOracle` functions are external and can be called without any restriction.

```
/**
 * @title Oracle Hub
 * @author Arcadia Finance
 * @notice The Oracle Hub stores the addresses and other necessary information of the Price Oracles and returns rates of
↪  assets
 * @dev No end-user should directly interact with the Oracle-Hub, only the Main Registry, Sub-Registries, or the
↪  contract owner.
 */

/**
 * @notice Sets an oracle to inactive if it has not been updated in the last week or if its answer is below the minimum
↪  answer.
 * @param oracle The address of the oracle to be checked
 * @dev An inactive oracle will always return a rate of 0.
 * @dev Anyone can call this function as part of an oracle failsafe mechanism.
 * Next to the deposit limits, the rate of an asset can be set to 0 if the oracle is not performing as intended.
 * @dev If the oracle becomes functional again (all checks pass), anyone can activate the oracle again.
 */
```

Recommendation: Consider modifying the documentation and removing wrong statements.

7.1.2 Incorrect inline documentation for `_IMPLEMENTATION_SLOT`

The inline documentation for the `_IMPLEMENTATION_SLOT` states that:

```
    /**
     * @dev Storage slot with the address of the current implementation.
     * This is the keccak-256 hash of "eip1967.proxy.implementation" subtracted by 1.
     */
```

However, in the constructor, there is no validation of this value.

Recommendation: Consider unifying documentation with code by changing documentation or code. Below we propose code for validation of the `_IMPLEMENTATION_SLOT` value:

```
assert(_IMPLEMENTATION_SLOT == bytes32(uint256(keccak256("eip1967.proxy.implementation")) - 1));
```

8 Test Suite Evaluation

8.1 Contracts Compilation Output

8.1.1 Arcadia Vault

```
$ forge compile
[] Compiling...
[] Compiling 153 files with 0.8.18
[] Solc 0.8.18 finished in 105.24s
Compiler run successful
```

8.1.2 Arcadia Lending

```
$ forge compile
[] Compiling...
[] Compiling 50 files with 0.8.18
[] Solc 0.8.18 finished in 16.30s
Compiler run successful
```

8.2 Tests Output

8.2.1 Arcadia Vault

```
$ forge test
[] Compiling...
No files changed, compilation skipped

Running 1 test for src/test/12_UniswapV2PricingModule.t.sol:DeploymentTest
Test result: ok. 1 passed; 0 failed; finished in 14.72ms

Running 1 test for src/test/gasTests/Liquidate3.t.sol:gasLiquidate_1ERC201ERC721
Test result: ok. 1 passed; 0 failed; finished in 19.46ms

Running 6 tests for src/test/gasTests/BuyVault4.t.sol:gasBuyVault_2ERC202ERC721
Test result: ok. 6 passed; 0 failed; finished in 23.33ms

Running 1 test for src/test/gasTests/Liquidate4.t.sol:gasLiquidate_2ERC202ERC721
Test result: ok. 1 passed; 0 failed; finished in 21.08ms

Running 7 tests for src/test/gasTests/Deploys.t.sol:gasDeploys
Test result: ok. 7 passed; 0 failed; finished in 24.50ms

Running 2 tests for src/test/gasTests/ProxyDepoy.sol:gasProxyDeploy
Test result: ok. 2 passed; 0 failed; finished in 22.65ms

Running 14 tests for src/test/gasTests/Deposits.t.sol:gasDeposits
Test result: ok. 14 passed; 0 failed; finished in 31.50ms

Running 3 tests for src/test/gasTests/Repay1.t.sol:gasRepay_1ERC20
Test result: ok. 3 passed; 0 failed; finished in 51.17ms

Running 3 tests for src/test/gasTests/Repay2.t.sol:gasRepay_2ERC20
Test result: ok. 3 passed; 0 failed; finished in 36.71ms

Running 3 tests for src/test/gasTests/Repay3.t.sol:gasRepay_1ERC201ERC721
Test result: ok. 3 passed; 0 failed; finished in 22.41ms

Running 3 tests for src/test/gasTests/Repay4.t.sol:gasRepay_2ERC202ERC721
Test result: ok. 3 passed; 0 failed; finished in 28.25ms

Running 9 tests for src/test/unit/Guardian.t.sol:BaseGuardianUnitTest
Test result: ok. 9 passed; 0 failed; finished in 147.66ms

Running 2 tests for src/test/unit/RiskModule.t.sol:RiskModuleTest
Test result: ok. 2 passed; 0 failed; finished in 70.13ms
```

```
Running 6 tests for src/test/gasTests/VaultAuction1.t.sol:gasVaultAuction_1ERC20
Test result: ok. 6 passed; 0 failed; finished in 26.91ms

Running 6 tests for src/test/gasTests/VaultAuction2.t.sol:gasVaultAuction_2ERC20
Test result: ok. 6 passed; 0 failed; finished in 26.12ms

Running 6 tests for src/test/unit/Guardian.t.sol:FactoryGuardianUnitTest
Test result: ok. 6 passed; 0 failed; finished in 118.99ms

Running 6 tests for src/test/gasTests/VaultAuction3.t.sol:gasVaultAuction_1ERC201ERC721
Test result: ok. 6 passed; 0 failed; finished in 26.13ms

Running 6 tests for src/test/gasTests/VaultAuction4.t.sol:gasVaultAuction_2ERC202ERC721
Test result: ok. 6 passed; 0 failed; finished in 28.15ms

Running 6 tests for src/test/gasTests/Withdrawal1.t.sol:gasWithdrawal1_1ERC20
Test result: ok. 6 passed; 0 failed; finished in 34.33ms

Running 12 tests for src/test/14_ATokenPricingModule.t.sol:aTokenPricingModuleTest
Test result: ok. 12 passed; 0 failed; finished in 454.01ms

Running 7 tests for src/test/gasTests/Withdrawal2.t.sol:gasWithdrawal2_2ERC20
Test result: ok. 7 passed; 0 failed; finished in 28.97ms

Running 6 tests for src/test/unit/Guardian.t.sol:MainRegistryGuardianUnitTest
Test result: ok. 6 passed; 0 failed; finished in 120.18ms

Running 1 test for src/test/gasTests/Liquidate1.t.sol:gasLiquidate_1ERC20
Test result: ok. 1 passed; 0 failed; finished in 26.30ms

Running 7 tests for src/test/gasTests/Withdrawal3.t.sol:gasWithdrawal3_1ERC201ERC721
Test result: ok. 7 passed; 0 failed; finished in 32.72ms

Running 8 tests for src/test/gasTests/Withdrawal4.t.sol:gasWithdrawal4_2ERC202ERC721
Test result: ok. 8 passed; 0 failed; finished in 22.93ms

Running 1 test for src/test/gasTests/Liquidate2.t.sol:gasLiquidate_2ERC20
Test result: ok. 1 passed; 0 failed; finished in 27.60ms

Running 5 tests for src/test/unit/ArcadiaOracleTest.t.sol:ArcadiaOracleTest
Test result: ok. 5 passed; 0 failed; finished in 3.56ms

Running 6 tests for src/test/gasTests/BuyVault1.sol:gasBuyVault_1ERC20
Test result: ok. 6 passed; 0 failed; finished in 37.75ms

Running 18 tests for src/test/6_MainRegistry.t.sol:AssetManagementTest
Test result: ok. 18 passed; 0 failed; finished in 603.80ms

Running 6 tests for src/test/gasTests/BuyVault2.sol:gasBuyVault_2ERC20
Test result: ok. 6 passed; 0 failed; finished in 51.65ms

Running 3 tests for src/test/6_MainRegistry.t.sol:BaseCurrencyManagementTest
Test result: ok. 3 passed; 0 failed; finished in 54.77ms

Running 6 tests for src/test/gasTests/BuyVault3.sol:gasBuyVault_1ERC201ERC721
Test result: ok. 6 passed; 0 failed; finished in 33.59ms

Running 7 tests for src/test/8_Vault.t.sol:VaultActionTest
Test result: ok. 7 passed; 0 failed; finished in 682.39ms

Running 2 tests for src/test/6_MainRegistry.t.sol:DeploymentTest
Test result: ok. 2 passed; 0 failed; finished in 27.57ms

Running 2 tests for src/test/6_MainRegistry.t.sol:ExternalContractsTest
Test result: ok. 2 passed; 0 failed; finished in 103.19ms

Running 3 tests for src/test/6_MainRegistry.t.sol:PriceModuleManagementTest
Test result: ok. 3 passed; 0 failed; finished in 50.55ms
```

```
Running 6 tests for src/test/8_Vault.t.sol:VaultManagementTest
Test result: ok. 6 passed; 0 failed; finished in 190.46ms

Running 11 tests for src/test/6_MainRegistry.t.sol:PricingLogicTest
Test result: ok. 11 passed; 0 failed; finished in 559.07ms

Running 12 tests for src/test/10_EndToEnd.t.sol:BorrowAndRepay
Test result: ok. 12 passed; 0 failed; finished in 1.99s

Running 6 tests for src/test/12_UniswapV2PricingModule.t.sol:AssetManagement
Test result: ok. 6 passed; 0 failed; finished in 72.84ms

Running 12 tests for src/test/15_StandardERC4626PricingModule.t.sol:standardERC4626PricingModuleTest
Test result: ok. 12 passed; 0 failed; finished in 1.88s

Running 2 tests for src/test/16_ActionMultiCall.t.sol:ActionMultiCallTest
Test result: ok. 2 passed; 0 failed; finished in 55.04ms

Running 17 tests for src/test/4_FloorERC721PricingModule.t.sol:FloorERC721PricingModuleTest
Test result: ok. 17 passed; 0 failed; finished in 445.15ms

Running 29 tests for src/test/1_OracleHub.t.sol:OracleHubTest
Test result: ok. 29 passed; 0 failed; finished in 2.00s

Running 30 tests for src/test/9_Liquidator.t.sol:LiquidatorTest
Test result: ok. 30 passed; 0 failed; finished in 3.55s

Running 1 test for src/test/8_Vault.t.sol:DeploymentTest
Test result: ok. 1 passed; 0 failed; finished in 44.19ms

Running 3 tests for src/test/8_Vault.t.sol:LiquidationLogicTest
Test result: ok. 3 passed; 0 failed; finished in 143.66ms

Running 9 tests for src/test/8_Vault.t.sol:MarginAccountSettingsTest
Test result: ok. 9 passed; 0 failed; finished in 184.14ms

Running 11 tests for src/test/12_UniswapV2PricingModule.t.sol:PricingLogic
Test result: ok. 11 passed; 0 failed; finished in 4.72s

Running 17 tests for src/test/5_FloorERC1155PricingModule.t.sol:FloorERC1155PricingModuleTest
Test result: ok. 17 passed; 0 failed; finished in 739.84ms

Running 4 tests for src/test/12_UniswapV2PricingModule.t.sol:UniswapV2Fees
Test result: ok. 4 passed; 0 failed; finished in 178.20ms

Running 3 tests for src/test/8_Vault.t.sol:OwnershipManagementTest
Test result: ok. 3 passed; 0 failed; finished in 112.77ms

Running 2 tests for src/test/10_EndToEnd.t.sol:DoActionWithLeverage
Test result: ok. 2 passed; 0 failed; finished in 5.19s

Running 4 tests for src/test/13_ProxyUpgrade.t.sol:VaultV2Test
Test result: ok. 4 passed; 0 failed; finished in 686.06ms

Running 8 tests for src/test/8_Vault.t.sol:DepreciatedTest
Test result: ok. 8 passed; 0 failed; finished in 1.46s

Running 11 tests for src/test/3_StandardERC20PricingModule.t.sol:StandardERC20PricingModuleTest
Test result: ok. 11 passed; 0 failed; finished in 362.40ms

Running 4 tests for src/test/8_Vault.t.sol:BaseCurrencyLogicTest
Test result: ok. 4 passed; 0 failed; finished in 123.60ms

Running 22 tests for src/test/2_AbstractPricingModule.t.sol:AbstractPricingModuleTest
Test result: ok. 22 passed; 0 failed; finished in 7.27s
```

8.2.2 Arcadia Lending

```
forge test
[] Compiling...
No files changed, compilation skipped

Running 1 test for test/DebtToken.t.sol:DeploymentTest
Test result: ok. 1 passed; 0 failed; finished in 28.19ms

Running 1 test for test/Tranche.t.sol:DeploymentTest
Test result: ok. 1 passed; 0 failed; finished in 27.84ms

Running 4 tests for test/DebtToken.t.sol:TransferTest
Test result: ok. 4 passed; 0 failed; finished in 296.22ms

Running 1 test for test/LendingPool.t.sol:DeploymentTest
Test result: ok. 1 passed; 0 failed; finished in 4.57ms

Running 11 tests for test/LendingPool.t.sol:TranchesTest
Test result: ok. 11 passed; 0 failed; finished in 462.82ms

Running 7 tests for test/LendingPool.t.sol:GuardianTest
Test result: ok. 7 passed; 0 failed; finished in 560.37ms

Running 2 tests for test/LendingPool.t.sol:AccountingTest
Test result: ok. 2 passed; 0 failed; finished in 754.82ms

Running 6 tests for test/LendingPool.t.sol:VaultTest
Test result: ok. 6 passed; 0 failed; finished in 391.56ms

Running 3 tests for test/LendingPool.t.sol:InterestRateTest
Test result: ok. 3 passed; 0 failed; finished in 424.31ms

Running 1 test for test/Tranche.t.sol:AccountingTest
Test result: ok. 1 passed; 0 failed; finished in 224.39ms

Running 15 tests for test/Guardian.t.sol:GuardianUnitTest
Test result: ok. 15 passed; 0 failed; finished in 1.46s

Running 3 tests for test/LendingPool.t.sol:InterestsTest
Test result: ok. 3 passed; 0 failed; finished in 668.82ms

Running 8 tests for test/DebtToken.t.sol:DepositWithdrawalTest
Test result: ok. 8 passed; 0 failed; finished in 1.76s

Running 5 tests for test/InterestRateModule.t.sol:InterestRateModuleTest
Test result: ok. 5 passed; 0 failed; finished in 356.68ms

Running 4 tests for test/LendingPool.t.sol:ProtocolCapTest
Test result: ok. 4 passed; 0 failed; finished in 213.12ms

Running 8 tests for test/LendingPool.t.sol:ProtocolFeeTest
Test result: ok. 8 passed; 0 failed; finished in 247.94ms

Running 6 tests for test/Tranche.t.sol:LockingTest
Test result: ok. 6 passed; 0 failed; finished in 160.08ms

Running 9 tests for test/LendingPool.t.sol:LeveragedActions
Test result: ok. 9 passed; 0 failed; finished in 2.18s

Running 23 tests for test/Tranche.t.sol:DepositAndWithdrawalTest
Test result: ok. 23 passed; 0 failed; finished in 3.28s

Running 16 tests for test/LendingPool.t.sol:DepositAndWithdrawalTest
Test result: ok. 16 passed; 0 failed; finished in 3.01s

Running 15 tests for test/LendingPool.t.sol:LiquidationTest
Test result: ok. 15 passed; 0 failed; finished in 2.56s
```

8.3 Code Coverage

```
forge coverage
```

The relevant output is presented below.

8.3.1 Arcadia Vault

```
| File                                                  | % Lines           | % Statements       | % Branches       | % Funcs          |
|-------------------------------------------------------|-------------------|--------------------|------------------|------------------|
| src/Factory.sol                                       | 94.64% (53/56)    | 95.08% (58/61)     | 88.46% (23/26)   | 87.50% (14/16)   |
| src/Liquidator.sol                                    | 84.62% (55/65)    | 85.29% (58/68)     | 83.33% (30/36)   | 90.00% (9/10)    |
| src/MainRegistry.sol                                  | 98.59% (70/71)    | 98.67% (74/75)     | 79.17% (19/24)   | 91.67% (11/12)   |
| src/OracleHub.sol                                     | 100.00% (38/38)   | 100.00% (40/40)    | 100.00% (28/28)  | 100.00% (4/4)    |
| src/PricingModules/AbstractPricingModule.sol          | 100.00% (23/23)   | 100.00% (27/27)    | 91.67% (11/12)   | 90.00% (9/10)    |
| src/PricingModules/StandardERC20PricingModule.sol     | 100.00% (20/20)   | 100.00% (21/21)    | 100.00% (6/6)    | 100.00% (3/3)    |
| src/Proxy.sol                                         | 100.00% (2/2)     | 100.00% (2/2)      | 100.00% (0/0)    | 100.00% (3/3)    |
| src/RiskModule.sol                                    | 100.00% (10/10)   | 100.00% (12/12)    | 100.00% (0/0)    | 100.00% (2/2)    |
| src/Vault.sol                                         | 93.75% (210/224)  | 93.75% (225/240)   | 73.21% (82/112)  | 97.06% (33/34)   |
| src/actions/ActionBase.sol                            | 100.00% (0/0)     | 100.00% (0/0)      | 100.00% (0/0)    | 0.00% (0/1)      |
| src/actions/MultiCall.sol                             | 86.67% (13/15)    | 88.89% (16/18)     | 50.00% (4/8)     | 100.00% (1/1)    |
| src/security/BaseGuardian.sol                         | 100.00% (2/2)     | 100.00% (2/2)      | 100.00% (0/0)    | 33.33% (1/3)     |
| src/security/FactoryGuardian.sol                      | 61.54% (8/13)     | 61.54% (8/13)      | 16.67% (1/6)     | 66.67% (2/3)     |
| src/security/MainRegistryGuardian.sol                 | 61.54% (8/13)     | 61.54% (8/13)      | 16.67% (1/6)     | 66.67% (2/3)     |
```

8.3.2 Arcadia Lending

```
| File                            | % Lines           | % Statements       | % Branches       | % Funcs          |
|---------------------------------|-------------------|--------------------|------------------|------------------|
| src/DebtToken.sol               | 100.00% (17/17)   | 100.00% (18/18)    | 100.00% (6/6)    | 90.91% (10/11)   |
| src/InterestRateModule.sol      | 100.00% (11/11)   | 100.00% (12/12)    | 75.00% (3/4)     | 100.00% (3/3)    |
| src/LendingPool.sol             | 100.00% (158/158) | 100.00% (171/171)  | 93.75% (60/64)   | 100.00% (33/33)  |
| src/Tranche.sol                 | 100.00% (42/42)   | 100.00% (42/42)    | 100.00% (16/16)  | 53.33% (8/15)    |
| src/TrustedCreditor.sol         | 100.00% (1/1)     | 100.00% (1/1)      | 100.00% (0/0)    | 100.00% (1/1)    |
| src/security/Guardian.sol       | 100.00% (24/24)   | 100.00% (24/24)    | 100.00% (6/6)    | 100.00% (4/4)    |
```

8.4 Slither

All the relevant issues raised by Slither have been incorporated into the issues described in this report.

9 About Nethermind

Founded in 2017 by a small team of world-class technologists, Nethermind builds Ethereum solutions for developers and enterprises. Boosted by a grant from the Ethereum Foundation in August 2018, our team has worked tirelessly to deliver the fastest Ethereum client in the market. Our flagship Ethereum client is all about performance and flexibility. Built on .NET core, a widespread, enterprise-friendly platform, Nethermind makes integration with existing infrastructures simple, without losing sight of stability, reliability, data integrity, and security. **Nethermind is made up of several engineering teams across various disciplines, all collaborating to realize the Ethereum roadmap, by conducting research and building high-quality tools**. Teams focus on specific areas of the Ethereum problem space. Each consists of specialists and experienced developers working alongside interns, learning the ropes in the Nethermind Internship Program. **Our mission is to gather passionate talent from around the world, and to tackle some of the blockchain's most complex problems**. Nethermind provides software solutions and services for developers and enterprises building the Ethereum ecosystem. We offer security reviews to projects built on EVM compatible chains and StarkNet. We have expertise in multiple areas of the Ethereum ecosystem, including protocol design, smart contracts (written in Solidity and Cairo), MEV, etc. We develop some of the most used tools on Starknet and one of the most used Ethereum clients. Learn more about us at `https://nethermind.io`.