



**PALADIN**  
BLOCKCHAIN SECURITY

# Smart Contract Security Assessment

Final Report

For Ambit Finance

19 November 2023



[paladinsec.co](https://paladinsec.co)



[info@paladinsec.co](mailto:info@paladinsec.co)

# Table of Contents

Table of Contents	2
Disclaimer	8
1 Overview	9
1.1 Summary	9
1.2 Contracts Assessed	10
1.3 Findings Summary	12
1.3.1 Global Issues	13
1.3.2 AddressRegistryExtensions	13
1.3.3 Errors	13
1.3.4 Fees	14
1.3.5 InterestMath	14
1.3.6 Normalizer	14
1.3.7 PercentageMath	14
1.3.8 PortfolioAssetExtensions	14
1.3.9 RayMath	15
1.3.10 USDMath	15
1.3.11 AddressRegistry	15
1.3.12 AssetStorage	15
1.3.13 Overseer	16
1.3.14 Treasury	16
1.3.15 DynamicInterestRateModel	16
1.3.16 FixedInterestRateModel	16
1.3.17 Liquidator	17
1.3.18 Market	18
1.3.19 MarketLiquidation	19
1.3.20 MarketStorage	19
1.3.21 DepositorVaultMarketplaceAdapter	19

1.3.22 Marketplace	20
1.3.23 SpotMarketMarketplaceAdapter	20
1.3.24 ChainlinkAggregatorPriceOracle	21
1.3.25 DepositorVaultTokenPriceOracle	21
1.3.26 FallbackPriceOracle	21
1.3.27 Custodian	22
1.3.28 CustodianMigrator	22
1.3.29 Portfolio	22
1.3.30 PortfolioStorage	23
1.3.31 AdminAccessControl	23
1.3.32 AuthorizedAccessControl	23
1.3.33 AccessControlList	23
1.3.34 AmbitToken	23
1.3.35 Pausable	24
1.3.36 Sweepable	24
1.3.37 DepositorVault	24
1.3.38 DepositorVaultMigrator	25
1.3.39 DepositorVaultStorage	25
1.3.40 DepositorVaultToken	25
1.3.41 LinearDistributedYieldVault	26
1.3.42 Vault	26
1.3.43 Flashlender	26
2 Findings	27
2.1 Global Issues	27
2.1.1 Issues & Recommendations	28
2.2 Libraries/AddressRegistryExtensions	34
2.2.1 Issues & Recommendations	34
2.3 Libraries/Errors	35
2.3.1 Issues & Recommendations	35
2.4 Libraries/Fees	36

2.4.1 Issues & Recommendations	36
2.5 Libraries/InterestMath	37
2.5.2 Issues & Recommendations	37
2.6 Libraries/Normalizer	38
2.6.1 Issues & Recommendations	39
2.7 Libraries/PercentageMath	40
2.7.1 Issues & Recommendations	40
2.8 Libraries/PortfolioAssetExtensions	41
2.8.1 Issues & Recommendations	41
2.9 Libraries/RayMath	42
2.9.1 Issues & Recommendations	43
2.10 Libraries/USDMath	45
2.10.1 Issues & Recommendations	45
2.11 Core/AddressRegistry	46
2.11.1 Privileged Functions	46
2.11.2 Issues & Recommendations	46
2.12 Core/AssetStorage	47
2.12.1 Privileged Functions	47
2.12.2 Issues & Recommendations	48
2.13 Governance/Overseer	52
2.13.1 Privileged Functions	52
2.13.2 Issues & Recommendations	53
2.14 Governance/Treasury	55
2.14.1 Privileged Functions	55
2.14.2 Issues & Recommendations	56
2.15 Market/DynamicInterestRateModel	57
2.15.1 Issues & Recommendations	58
2.16 Market/FixedInterestRateModel	60
2.16.1 Privileged Functions	60
2.16.2 Issues & Recommendations	61

2.17 Market/Liquidator	63
2.17.1 Privileged Functions	63
2.17.2 Issues & Recommendations	64
2.18 Market/Market	72
2.18.1 Privileged Functions	72
2.18.2 Issues & Recommendations	73
2.19 Market/MarketLiquidation	84
2.19.1 Privileged Functions	85
2.19.2 Issues & Recommendations	86
2.20 Market/MarketStorage	95
2.20.1 Privileged Functions	95
2.20.2 Issues & Recommendations	95
2.21 Marketplace/DepositorVaultMarketplaceAdapter	96
2.21.1 Issues & Recommendations	97
2.22 Marketplace/Marketplace	99
2.22.1 Issues & Recommendations	100
2.23 Marketplace/SpotMarketMarketplaceAdapter	110
2.23.1 Issues & Recommendations	111
2.24 Oracle/ChainlinkAggregatorPriceOracle	117
2.24.1 Issues & Recommendations	117
2.25 Oracle/DepositorVaultTokenPriceOracle	118
2.25.1 Issues & Recommendations	119
2.26 Oracle/FallbackPriceOracle	122
2.26.1 Issues & Recommendations	123
2.27 Portfolio/Custodian	126
2.27.1 Privileged Functions	126
2.27.2 Issues & Recommendations	127
2.28 Portfolio/CustodianMigrator	130
2.28.1 Issues & Recommendations	130
2.29 Portfolio/Portfolio	131

2.29.1 Privileged Functions	131
2.29.2 Issues & Recommendations	132
2.30 Portfolio/PortfolioStorage	138
2.30.1 Privileged Functions	138
2.30.2 Issues & Recommendations	138
2.31 Security/AdminAccessControl	139
2.31.1 Privileged Functions	139
2.31.2 Issues & Recommendations	140
2.32 Security/AuthorizedAccessControl	141
2.32.1 Issues & Recommendations	141
2.33 Security/AccessControlList	142
2.33.1 Issues & Recommendations	142
2.34 Tokens/AmbitToken	143
2.34.1 Privileged Functions	143
2.34.2 Issues & Recommendations	144
2.35 Utils/Pausable	145
2.35.1 Privileged Functions	145
2.35.2 Issues & Recommendations	145
2.36 Utils/Sweepable	146
2.36.1 Privileged Functions	146
2.36.2 Issues & Recommendations	147
2.37 Vault/DepositorVault	149
2.37.1 Privileged Functions	150
2.37.2 Issues & Recommendations	151
2.38 Vault/DepositorVaultMigrator	166
2.38.1 Privileged Functions	166
2.38.2 Issues & Recommendations	167
2.39 Vault/DepositorVaultStorage	171
2.39.1 Privileged Functions	171
2.39.2 Issues & Recommendations	171

2.40 Vault/DepositorVaultToken	172
2.40.1 Privileged Functions	172
2.40.2 Issues & Recommendations	172
2.41 Vault/LinearDistributedYieldVault	173
2.41.1 Issues & Recommendations	174
2.42 Vault/Vault	178
2.42.1 Issues & Recommendations	179
2.42 Vault/Flashlender	182
2.42.1 Issues & Recommendations	182



# Disclaimer

Paladin Blockchain Security ("Paladin") has conducted an independent audit to verify the integrity of and highlight any vulnerabilities or errors, intentional or unintentional, that may be present in the codes that were provided for the scope of this audit. This audit report does not constitute agreement, acceptance or advocacy for the Project that was audited, and users relying on this audit report should not consider this as having any merit for financial advice in any shape, form or nature. The contracts audited do not account for any economic developments that may be pursued by the Project in question, and that the veracity of the findings thus presented in this report relate solely to the proficiency, competence, aptitude and discretion of our independent auditors, who make no guarantees nor assurance that the contracts are completely free of exploits, bugs, vulnerabilities or deprecation of technologies. Further, this audit report shall not be disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by Paladin.

All information provided in this report does not constitute financial or investment advice, nor should it be used to signal that any persons reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report. Information is provided 'as is', and Paladin is under no covenant to the completeness, accuracy or solidity of the contracts audited. In no event will Paladin or its partners, employees, agents or parties related to the provision of this audit report be liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this audit report.

Cryptocurrencies and any technologies by extension directly or indirectly related to cryptocurrencies are highly volatile and speculative by nature. All reasonable due diligence and safeguards may yet be insufficient, and users should exercise considerable caution when participating in any shape or form in this nascent industry.

The audit report has made all reasonable attempts to provide clear and articulate recommendations to the Project team with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts provided. It is the sole responsibility of the Project team to sufficiently test and perform checks, ensuring that the contracts are functioning as intended, specifically that the functions therein contained within said contracts have the desired intended effects, functionalities and outcomes of the Project team.

Paladin retains the right to re-use any and all knowledge and expertise gained during the audit process, including, but not limited to, vulnerabilities, bugs, or new attack vectors. Paladin is therefore allowed and expected to use this knowledge in subsequent audits and to inform any third party, who may or may not be our past or current clients, whose projects have similar vulnerabilities. Paladin is furthermore allowed to claim bug bounties from third-parties while doing so.



# 1 Overview

This report has been prepared for Ambit Finance on the Arbitrum network. Paladin provides a user-centred examination of the smart contracts to look for vulnerabilities, logic errors or other issues from both an internal and external perspective.

## 1.1 Summary

<b>Project Name</b>	Ambit Finance
<b>URL</b>	TBC
<b>Platform</b>	Arbitrum
<b>Language</b>	Solidity
<b>Preliminary Contracts</b>	<a href="https://github.com/ambitfi/ambitfi-contracts/tree/d0e992d817afc4fd2636505142e614e8b98a8b5d/contracts">https://github.com/ambitfi/ambitfi-contracts/tree/d0e992d817afc4fd2636505142e614e8b98a8b5d/contracts</a>
<b>Resolution 1</b>	<a href="https://github.com/ambitfi/ambitfi-contracts/tree/142bf2677ccb3a36eee709d8601e5532aa5df080/contracts">https://github.com/ambitfi/ambitfi-contracts/tree/142bf2677ccb3a36eee709d8601e5532aa5df080/contracts</a>
<b>Resolution 2</b>	<a href="https://github.com/ambitfi/ambitfi-contracts/tree/b8526ecb2b1ab27668a2c3f30d1aa4321040f94e/contracts">https://github.com/ambitfi/ambitfi-contracts/tree/b8526ecb2b1ab27668a2c3f30d1aa4321040f94e/contracts</a>

## 1.2 Contracts Assessed

Name	Contract	Live Code Match
AddressRegistryExtensions		
Errors		
Fees		
InterestMath		
Normalizer		
PercentageMath		
PortfolioAssetExtensions		
RayMath		
USDMath		
AddressRegistry		
AssetStorage		
Overseer		
Treasury		
DynamicInterestRateModel		
FixedInterestRateModel		
Liquidator		
Market		
MarketLiquidation		
MarketStorage		
DepositorVaultMarketplaceAdapter		

---

Marketplace

---

SpotMarketMarketpla  
ceAdapter

---

ChainlinkAggregator  
PriceOracle

---

DepositorVaultToken  
PriceOracle

---

FallbackPriceOracle

---

Custodian

---

CustodianMigrator

---

Portfolio

---

PortfolioStorage

---

AdminAccessControl

---

AuthorizedAccessCon  
trol

---

AccessControlList

---

AmbitToken

---

Pausable

---

Sweepable

---

DepositorVault

---

DepositorVaultMigra  
tor

---

DepositorVaultStora  
ge

---

DepositorVaultToken

---

LinearDistributedYi  
eldVault

---

Vault

---

Flashlender

---

## 1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
● Governance	1	-	-	1
● High	14	10	-	4
● Medium	18	12	2	4
● Low	49	27	1	21
● Informational	52	42	5	5
<b>Total</b>	<b>134</b>	<b>91</b>	<b>8</b>	<b>35</b>

### Classification of Issues

Severity	Description
● Governance	Issues under this category are where the governance or owners of the protocol have certain privileges that users need to be aware of, some of which can result in the loss of user funds if the governance's private keys are lost or if they turn malicious, for example.
● High	Exploits, vulnerabilities or errors that will certainly or probabilistically lead towards loss of funds, control, or impairment of the contract and its functions. Issues under this classification are recommended to be fixed with utmost urgency.
● Medium	Bugs or issues with that may be subject to exploit, though their impact is somewhat limited. Issues under this classification are recommended to be fixed as soon as possible.
● Low	Effects are minimal in isolation and do not pose a significant danger to the project or its users. Issues under this classification are recommended to be fixed nonetheless.
● Informational	Consistency, syntax or style best practices. Generally pose a negligible level of risk, if any.

## 1.3.1 Global Issues

ID	Severity	Summary	Status
01	GOV	Governance risk: Change of _interestRateModel can break contract	ACKNOWLEDGED
02	HIGH	General assumption of baseAsset being worth 1 USD	ACKNOWLEDGED
03	HIGH	Governance has full control over all funds in the ecosystem	ACKNOWLEDGED
04	LOW	Lack of staleness check when using Chainlink oracle	✓ RESOLVED
05	LOW	Lack of upper limit in portfolio assets	✓ RESOLVED
06	LOW	Pausing and unpausing can prevent/enable liquidations	✓ RESOLVED
07	INFO	Protocol does not work with tokens that have a fee on transfer	ACKNOWLEDGED
08	INFO	Sensitive structure for privileges	✓ RESOLVED
09	INFO	Users can supply and withdraw in the same block	✓ RESOLVED

## 1.3.2 AddressRegistryExtensions

ID	Severity	Summary	Status
10	INFO	Typographical issues	✓ RESOLVED

## 1.3.3 Errors

ID	Severity	Summary	Status
11	INFO	Unused custom errors	✓ RESOLVED

## 1.3.4 Fees

ID	Severity	Summary	Status
12	LOW	Lack of validation within used contracts	PARTIAL

## 1.3.5 InterestMath

ID	Severity	Summary	Status
13	INFO	Typographical issues	RESOLVED

## 1.3.6 Normalizer

ID	Severity	Summary	Status
14	LOW	Decimals will be truncated when converting to lower decimals	ACKNOWLEDGED

## 1.3.7 PercentageMath

ID	Severity	Summary	Status
15	INFO	Typographical issues	RESOLVED

## 1.3.8 PortfolioAssetExtensions

No issues found.

## 1.3.9 RayMath

ID	Severity	Summary	Status
16	LOW	add can overflow	✓ RESOLVED
17	INFO	Typographical issues	✓ RESOLVED

## 1.3.10 USDMath

ID	Severity	Summary	Status
18	INFO	Typographical issues	✓ RESOLVED

## 1.3.11 AddressRegistry

No issues found.

## 1.3.12 AssetStorage

ID	Severity	Summary	Status
19	MEDIUM	Adjustment of LTV can make users vulnerable to liquidation	✓ RESOLVED
20	MEDIUM	Lack of reasonable liquidationDiscount limit	ACKNOWLEDGED
21	LOW	getAssets can run out of gas	ACKNOWLEDGED
22	INFO	Gas optimizations	✓ RESOLVED
23	INFO	Typographical issues	✓ RESOLVED

### 1.3.13 Overseer

ID	Severity	Summary	Status
24	MEDIUM	shutdown functionality might run out of gas	✓ RESOLVED
25	LOW	Pausing functionality might lead to undesired side-effects	✓ RESOLVED
26	INFO	Typographical issues	✓ RESOLVED

### 1.3.14 Treasury

ID	Severity	Summary	Status
27	INFO	Typographical issues	✓ RESOLVED

### 1.3.15 DynamicInterestRateModel

ID	Severity	Summary	Status
28	MEDIUM	Interest rate mechanism is manipulatable due to utilization rate changes not always triggering an interest rate indexation	PARTIAL
29	INFO	Typographical issues	✓ RESOLVED

### 1.3.16 FixedInterestRateModel

ID	Severity	Summary	Status
30	MEDIUM	Update of interest rate will change borrowIndex retroactively	✓ RESOLVED
31	LOW	_rate can be set arbitrarily high	✓ RESOLVED
32	INFO	Typographical issues	✓ RESOLVED



## 1.3.17 Liquidator

ID	Severity	Summary	Status
33	HIGH	liquidateVaultToken function is fundamentally flawed, preventing any amUSDT token liquidations from occurring	RESOLVED
34	HIGH	Very large positions can become hard/impossible to liquidate	ACKNOWLEDGED
35	LOW	Liquidation of vaultToken can be prevented	ACKNOWLEDGED
36	LOW	Profit of vaultToken liquidation can be reduced	ACKNOWLEDGED
37	LOW	Allowing for unprofitable redemptions of vaultTokens is considered a risk	RESOLVED
38	LOW	Liquidation will not work if _baseAsset is not a USD stablecoin	ACKNOWLEDGED
39	LOW	Missing safeguard within transferFees	RESOLVED
40	INFO	Gas optimizations	RESOLVED
41	INFO	Typographical issues	PARTIAL
42	INFO	Slippage calculation is completely useless	RESOLVED

## 1.3.18 Market

ID	Severity	Summary	Status
43	HIGH	Stablecoin depositor vault can be fully drained if an asset with 20 decimals or more is ever added as collateral due to an overflow vulnerability in the borrowing limit calculation	RESOLVED
44	HIGH	ensureHealthyAccount returns true for a score of 100	RESOLVED
45	LOW	Borrowing can result in liquidation	RESOLVED
46	LOW	Distributing interest only at the end of the loan can lead to misalignment for long loan durations and even exploitation to reduce interest paid in certain edge cases	ACKNOWLEDGED
47	LOW	Automatic reduction of the repay amount to outstanding liabilities could cause issues for integrating applications who are unaware of this	ACKNOWLEDGED
48	LOW	Governance risk: _borrowingFee lacks upper limit	RESOLVED
49	INFO	Market state is calculated if no time has elapsed	RESOLVED
50	INFO	Gas optimizations	PARTIAL
51	INFO	Typographical issues	PARTIAL

## 1.3.19 MarketLiquidation

ID	Severity	Summary	Status
52	LOW	findLargestPosition does not include collateralFactor	ACKNOWLEDGED
53	LOW	calculateLiquidation potentially compares values in different denominations	RESOLVED
54	LOW	Funds are withdrawn directly from the Custodian instead of the PortfolioBroker	RESOLVED
55	LOW	requireUnhealthyAccount passes if liabilities are zero	RESOLVED
56	LOW	Highest position is potentially truncated	ACKNOWLEDGED
57	LOW	Potential mismatch between maxAmount and liquidationSupply	ACKNOWLEDGED
58	LOW	Incorrect decimal denomination for minimum comparison	RESOLVED
59	LOW	Small positions can never be fully liquidated	RESOLVED
60	LOW	Liquidation functions lack a minimum received slippage check for manual liquidations	ACKNOWLEDGED
60	INFO	Typographical issues	PARTIAL

## 1.3.20 MarketStorage

ID	Severity	Summary	Status
61	INFO	Typographical issues	RESOLVED

## 1.3.21 DepositorVaultMarketplaceAdapter

ID	Severity	Summary	Status
62	LOW	normalize can truncate price if vaultToken has less than 8 decimals	RESOLVED
63	INFO	Typographical issues	RESOLVED

## 1.3.22 Marketplace

ID	Severity	Summary	Status
64	HIGH	The brokerage functions of the marketplace contain faulty internal states, allowing for any reentrant behavior to potentially exploit the user and force liquidation	✓ RESOLVED
65	MEDIUM	bor rowFeeAmount is not accounted into slippage	✓ RESOLVED
66	MEDIUM	Healthy state is not guaranteed	✓ RESOLVED
67	LOW	available calculation does not work for baseAssets that are not USDT	✓ RESOLVED
68	LOW	Estimation of maxAmount does not account for fee nor slippage	ACKNOWLEDGED
69	LOW	Invalid logic if context . amount < context . maxAmount	✓ RESOLVED
70	LOW	sell does not return the total amountOut; instead, it returns the amountOut after the liabilities repaid have been deducted	✓ RESOLVED
71	INFO	Missing check that marketplaceAdapter is set when buying from your portfolio	✓ RESOLVED
72	INFO	Gas optimizations	✓ RESOLVED
73	INFO	Typographical issues	✓ RESOLVED

## 1.3.23 SpotMarketMarketplaceAdapter

ID	Severity	Summary	Status
74	HIGH	minAmountOut calculation for buy function normalizes incorrect decimals	✓ RESOLVED
75	MEDIUM	Inefficient architecture	ACKNOWLEDGED
76	MEDIUM	Truncation of decimals will result in price decrease	✓ RESOLVED
77	MEDIUM	Slippage application will result in higher slippage than expected	✓ RESOLVED
78	LOW	Hardcoded swapFee of 0.3%	ACKNOWLEDGED
79	INFO	Typographical issues	✓ RESOLVED

## 1.3.24 ChainlinkAggregatorPriceOracle

ID	Severity	Summary	Status
80	INFO	Typographical issues	✓ RESOLVED

## 1.3.25 DepositorVaultTokenPriceOracle

ID	Severity	Summary	Status
81	LOW	Lack of staleness check for denominator oracle	✓ RESOLVED
82	LOW	DepositorVaultTokenPriceOracle reports an incorrect price if the underlying token has a different number of decimals compared to the vault token	ACKNOWLEDGED
83	LOW	Price is truncated	ACKNOWLEDGED
84	INFO	Gas optimizations	✓ RESOLVED
85	INFO	Typographical issues	✓ RESOLVED

## 1.3.26 FallbackPriceOracle

ID	Severity	Summary	Status
86	LOW	Oracle can be made magnitudes more secure within the current design by allowing the price to only slowly go up but go down instantly	ACKNOWLEDGED
87	LOW	Staleness protection not guaranteed	ACKNOWLEDGED
88	INFO	isStale could theoretically underflow and revert	✓ RESOLVED
89	INFO	Gas optimizations	✓ RESOLVED
90	INFO	Typographical issues	✓ RESOLVED

## 1.3.27 Custodian

ID	Severity	Summary	Status
91	LOW	Checks-effects-interactions pattern is not adhered to, allows for the Portfolio maxSupply check to be bypassed through a reentrancy exploit	RESOLVED
92	INFO	Typographical issues	PARTIAL
93	INFO	Gas optimizations	ACKNOWLEDGED

## 1.3.28 CustodianMigrator

No issues found.

## 1.3.29 Portfolio

ID	Severity	Summary	Status
94	HIGH	ETH liquidation is impossible due to special handling of ETH compared to WETH	RESOLVED
95	MEDIUM	Custodian is called before share amount is updated	RESOLVED
96	LOW	Missing non-zero modifier	RESOLVED
97	LOW	Off-by-one error within supplyTo	RESOLVED
98	LOW	Inconsistency between share increase/decrease	RESOLVED
99	INFO	Additional layer of security: USD based caps	ACKNOWLEDGED
100	INFO	Typographical issues	RESOLVED
101	INFO	Gas optimizations	RESOLVED

### 1.3.30 PortfolioStorage

ID	Severity	Summary	Status
102	INFO	Gas optimizations	RESOLVED

### 1.3.31 AdminAccessControl

ID	Severity	Summary	Status
103	INFO	Unused import	RESOLVED

### 1.3.32 AuthorizedAccessControl

No issues found.

### 1.3.33 AccessControlList

No issues found.

### 1.3.34 AmbitToken

ID	Severity	Summary	Status
104	INFO	Typographical issues	ACKNOWLEDGED

### 1.3.35 Pausable

No issues found.

### 1.3.36 Sweepable

ID	Severity	Summary	Status
105	LOW	Sweeping Ether to treasury might not work for treasuries that execute logic on their fallback	✓ RESOLVED
106	INFO	Typographical issue	✓ RESOLVED
107	INFO	sweepETH is payable	✓ RESOLVED

### 1.3.37 DepositorVault

ID	Severity	Summary	Status
108	HIGH	DoS exploit: Several functions including <code>getAvailableBalance</code> can be exploited when utilization is 100% by withdrawing manually sent USDT, causing these functions to brick and revert due to an underflow vulnerability	✓ RESOLVED
109	HIGH	External users can manipulate the utilization ratio	ACKNOWLEDGED
110	HIGH	Users can potentially lose tokens during repay	✓ RESOLVED
111	MEDIUM	Flashloan fee will be stuck in the contract	✓ RESOLVED
112	MEDIUM	Reserve logic is redundant and can be circumvented	PARTIAL
113	MEDIUM	Potential reentrancy attack due to un-updated contract state	✓ RESOLVED
114	MEDIUM	Change of <code>borrowLimit</code> will change market state retroactively	ACKNOWLEDGED
115	MEDIUM	Architectural flaw: Initial positions will not receive any yield	ACKNOWLEDGED
116	LOW	Uncapped flashloan fee	✓ RESOLVED
117	LOW	<code>borrowLimit</code> can be set to be less than liabilities	ACKNOWLEDGED
118	LOW	<code>getBorrowerUtilization</code> will return <code>1e27</code> for limit of zero	✓ RESOLVED
119	LOW	Lack of checks-effects-interactions allows the <code>maxSupply</code> check to be bypassed if the token is vulnerable to re-entrancy	✓ RESOLVED



120	LOW	The reserve assets cannot be flashloaned	✓ RESOLVED
121	INFO	Gas optimizations	✓ RESOLVED
122	INFO	Typographical issues	✓ RESOLVED

### 1.3.38 DepositorVaultMigrator

ID	Severity	Summary	Status
123	HIGH	Yield will be lost with new vault	✓ RESOLVED
124	MEDIUM	Several variables are unset after deployment	✓ RESOLVED
125	MEDIUM	Swept funds are sent to the wrong vault	✓ RESOLVED
126	INFO	Typographical issues	✓ RESOLVED
127	INFO	Vendor and Purchaser are still granted rights	✓ RESOLVED

### 1.3.39 DepositorVaultStorage

No issues found.

### 1.3.40 DepositorVaultToken

No issues found.



### 1.3.41 LinearDistributedYieldVault

ID	Severity	Summary	Status
128	LOW	Non-configurable <code>_distributionWindow</code> may backfire in the future	ACKNOWLEDGED
129	LOW	The distribution rate is counter-intuitively non-linear depending on when <code>distributeYield</code> gets called	ACKNOWLEDGED
130	INFO	Lack of checks-effects-interactions	✓ RESOLVED
131	INFO	Gas optimizations	✓ RESOLVED
132	INFO	Typographical issues	ACKNOWLEDGED

### 1.3.42 Vault

ID	Severity	Summary	Status
133	HIGH	Share distribution can be manipulated by first depositor	✓ RESOLVED
134	INFO	Gas optimizations	✓ RESOLVED

### 1.3.43 Flashlender

No issues found.



# 2 Findings

---

## 2.1 Global Issues

The issues listed in this section apply to the protocol as a whole. Please read through them carefully and take care to apply the fixes across the relevant contracts.



## 2.1.1 Issues & Recommendations

<b>Issue #01</b>	<b>Governance risk: Change of <code>_interestRateModel</code> can break contract</b>
<b>Severity</b>	<span>● GOVERNANCE</span>
<b>Description</b>	The admin can change <code>_interestRateModel</code> . This address is called within <code>calculateMarketState</code> and can therefore DoS several functionalities, including <code>accrueLiabilities</code> .
<b>Recommendation</b>	Consider keeping the admin address strictly under a multi-signature wallet with KYC-ed participants.
<b>Resolution</b>	<span>● ACKNOWLEDGED</span>

<b>Issue #02</b>	<b>General assumption of <code>baseAsset</code> being worth 1 USD</b>
<b>Severity</b>	<span>● HIGH SEVERITY</span>
<b>Description</b>	<p>Throughout the codebase, there are several value comparisons between the <code>borrowLimit</code> and the current liabilities, where the <code>borrowLimit</code> is always denominated in USD with 8 -&gt; 18 decimals.</p> <p>This will result in issues if the <code>baseAsset</code> should ever be worth less than 1 USD resulting in potentially undesirable liquidations, since the debt position is practically worth less than 1 USD but can still be liquidated.</p>
<b>Recommendation</b>	Consider keeping this scenario and mind and brain-storming potential emergency scenarios in case such a black-swan event occurs.
<b>Resolution</b>	<span>● ACKNOWLEDGED</span>

**Issue #03****Governance has full control over all funds in the ecosystem****Severity** HIGH SEVERITY**Description**

The whole architecture is built in such a manner that governance can affect different state transitions, apply changes to important state variables which can result in a loss of funds in various different ways. Furthermore, many of the open approvals can be drained due to broker functions directly pulling funds from the accounts.


Users should therefore be extremely diligent in only approving the number of coins that they actually plan to use within the system, as open approvals can and will be stolen in case the system becomes compromised (keys get stolen).

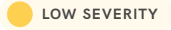

**Update:** In the resolution round, several contracts have become migrate-able, granting admins several other privileges that can potentially lead to loss of user funds.



**Recommendation**

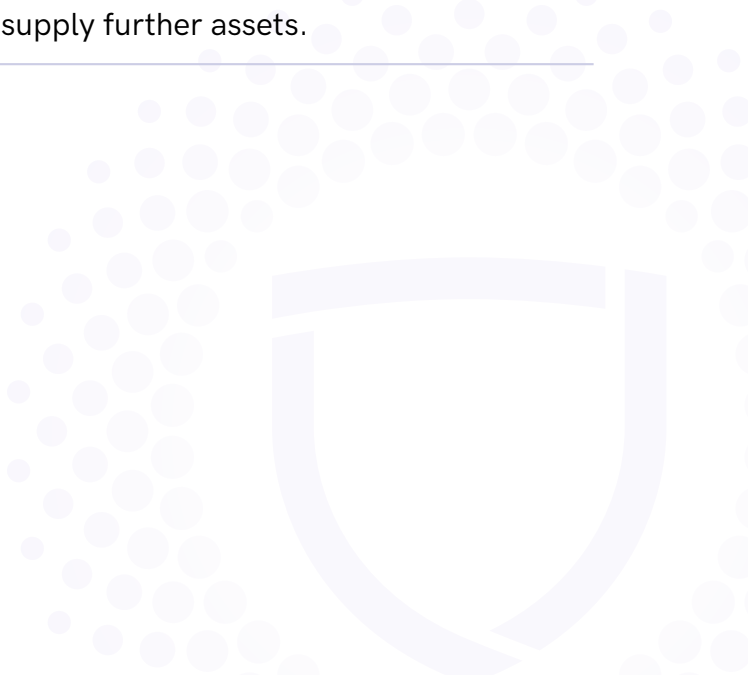
Consider only using multi-signature wallets with KYC-ed participants for privileged roles within the whole architecture.

Consider redesigning the `safeTransferFrom` calls within broker and Custodian to pull from `msg.sender` instead of the original accounts. This requires users to explicitly approve the broker they trust instead of having to approve potentially malicious brokers that can pull from their account.

**Resolution** ACKNOWLEDGED

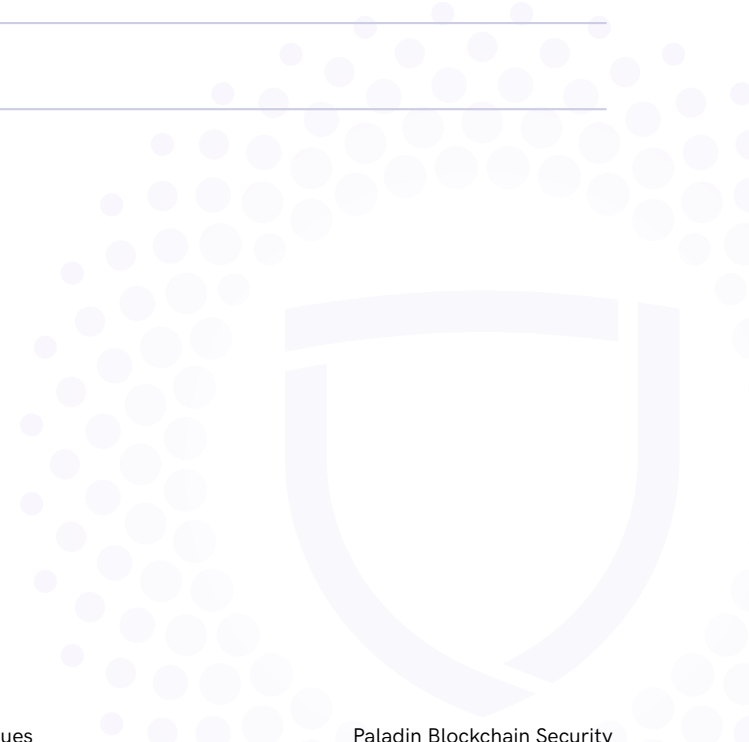
<b>Issue #04</b>	<b>Lack of staleness check when using Chainlink oracle</b>
<b>Severity</b>	 LOW SEVERITY
<b>Description</b>	There is a lack of staleness check whenever the Chainlink oracle is used. This can lead to incorrect return values, resulting in a loss of funds.
<b>Recommendation</b>	Consider implementing a staleness check — this can also be done directly in the ChainlinkAggregatorPriceOracle contract instead of on every single implementation.
<b>Resolution</b>	 RESOLVED

<b>Issue #05</b>	<b>Lack of upper limit in portfolio assets</b>
<b>Severity</b>	 LOW SEVERITY
<b>Description</b>	The contract has several sections which potentially loop over all existing assets. There is a risk that these loops run out of gas, potentially preventing liquidations and DoS'ing further functionalities.
<b>Recommendation</b>	Consider setting a reasonable upper limit to how many assets can be added.
<b>Resolution</b>	 RESOLVED However, a bug was introduced — once MAX_PORTFOLIO_ASSETS is reached, there is no way to supply further assets.



<b>Issue #06</b>	<b>Pausing and unpausing can prevent/enable liquidations</b>
<b>Severity</b>	<span style="color: yellow;">●</span> LOW SEVERITY
<b>Location</b>	<pre>function repay(uint256 amount) external whenNotPaused  function repayOutstandingLiabilities() external whenNotPaused</pre>
<b>Description</b>	The repay and repayOutstandingLiabilities functions have a whenNotPaused modifier in the vault and market contracts that will prevent users from repaying their liabilities if set to paused.
<b>Recommendation</b>	Consider removing the whenNotPaused modifier of repay() and repayOutstandingLiabilities().
<b>Resolution</b>	<span style="color: green;">✓</span> RESOLVED Repayments are now allowed when the project is paused.

<b>Issue #07</b>	<b>Protocol does not work with tokens that have a fee on transfer</b>
<b>Severity</b>	<span style="color: purple;">●</span> INFORMATIONAL
<b>Description</b>	There are several sections within the protocol architecture that will not work with tokens with a fee on transfer — this will break the whole system.
<b>Recommendation</b>	Consider not using such tokens.
<b>Resolution</b>	<span style="color: grey;">●</span> ACKNOWLEDGED



**Issue #08****Sensitive structure for privileges****Severity** INFORMATIONAL**Description**

Majority of the contracts inherit either the `AdminAccessControl` or `AuthorizedAccessControl` contract, whereas the deployer will always get the default admin role granted.

It is of utmost importance that the setting for the privileged addresses is done carefully, otherwise functionalities might break or unprivileged third-parties can execute sensitive calls and storage variable changes.

**Recommendation**

Consider following `Permissions.md` with utmost care and eventually revoking the default admin role from the contract deployer.

**Resolution** RESOLVED

A new governance mechanism has been implemented.





**Issue #09****Users can supply and withdraw in the same block****Severity** INFORMATIONAL**Description**

As supplying and withdrawing directly are not functionalities that users with good intentions would use together, rather they would be used for malicious behavior, it might be desired to prevent users from supplying and withdrawing in the same block.

**Recommendation**

Consider adding a cooldown period between the supply and withdraw functions. However, this requires careful testing with all other functionalities in mind, to ensure the architecture still works as expected.

This might be considered inadequate, even internally within the Paladin team such solutions raise strong doubts. A stronger solution may be to be extremely careful with privileges granted to the user.

Since checks-effects-interactions is not adhered to everywhere, we strongly suggest re-writing the codebase to adhere to checks-effects-interactions to prevent reentrancy exploits (even though the client indicates no reentrancy tokens/routes will be supported). If not, adding a global reentrancy guard to prevent reentrancy risks can be considered adequate as well.

Additionally, taking extreme care that flash-attacks such as interest rate manipulation are simply not possible is also desired. This must be done on a case-by-case basis.

**Resolution** RESOLVED

---

## 2.2 Libraries/AddressRegistryExtensions

AddressRegistryExtensions is a simple registry contract that stores different variables as bytes32 hash. This library exposes several getter functions which internally call the getAddress functionality using a valid hash.

It is used in various contracts throughout the codebase whenever the AddressRegistry contract is intended to be called.

### 2.2.1 Issues & Recommendations

<b>Issue #10</b>	<b>Typographical issues</b>
<b>Severity</b>	<span>INFORMATIONAL</span>
<b>Description</b>	<p><u>L16</u></p> <pre>import { ILiquidator } from "../interfaces/market/ILiquidator.sol";</pre> <p>This import is unused and can therefore be removed</p> <p><u>L40</u></p> <pre>function getAmbitToken(IAddressRegistry self) internal view returns (address) {</pre> <p>It may make sense to return the token interface instead.</p>
<b>Recommendation</b>	Consider fixing the typographical issues.
<b>Resolution</b>	<span>RESOLVED</span>

---

## 2.3 Libraries/Errors

Errors is a simple registry of all custom revert errors.

### 2.3.1 Issues & Recommendations

<b>Issue #11</b>	<b>Unused custom errors</b>
<b>Severity</b>	<span>INFORMATIONAL</span>
<b>Description</b>	<p><u>L30</u> error Marketplace_StalePrice(uint age);</p> <p><u>L31</u> error Marketplace_WithdrawAmountTooLarge(uint256 amount, uint256 shares, uint256 totalShares);</p> <p><u>L63</u> error DynamicInterestRateModel_PrecisionTooLarge(uint256 decimals);</p>
<b>Description</b>	The contract contains errors which are unused within the architecture.
<b>Recommendation</b>	Consider either removing or using these errors.
<b>Resolution</b>	<span>RESOLVED</span> However, with the introduction of new errors, the following error remains unused: Portfolio_SlippageExceeded

---

## 2.4 Libraries/Fees

Fees is responsible for the fee calculation based on a given amount and bps, including a minimum comparison with the `maxAmount` of the `Parameters` struct. This library is used throughout the whole architecture.

### 2.4.1 Issues & Recommendations

<b>Issue #12</b>	<b>Lack of validation within used contracts</b>
<b>Severity</b>	<span>● LOW SEVERITY</span>
<b>Description</b>	Wherever the <code>Fees.Parameters</code> value is used and can be set, there is no upper limit for neither <code>bps</code> nor <code>maxAmount</code> . This can lead to undesired side-effects if these values are accidentally incorrectly set.
<b>Recommendation</b>	Consider implementing a reasonable upper limit wherever this struct is used.
<b>Resolution</b>	<span>● PARTIALLY RESOLVED</span> Within the <code>Liquidator</code> contract, it is ensured that both fees together cannot be larger than 100%, however, this still seems unreasonable in our opinion.



---

## 2.5 Libraries/InterestMath

InterestMath is used within the Market contract with the task of calculating the correct updated borrowIndex as well as calculating user liabilities based on the current borrowIndex.

The increase of borrowIndex depends on the time elapsed since the last update as well as the current rate, which is handled within the DynamicInterestRateModel.

### 2.5.2 Issues & Recommendations

<b>Issue #13</b>	<b>Typographical issues</b>
<b>Severity</b>	 INFORMATIONAL
<b>Description</b>	"elapsed" is misspelled as "ellapsed" throughout the contract.
<b>Recommendation</b>	Consider fixing the typographical issues.
<b>Resolution</b>	 RESOLVED



---

## 2.6 Libraries/Normalizer

Normalizer is used throughout the project's contracts and is used to convert amounts from their origin decimals to specific destination decimals. For example, if the number 1000 was converted from 3 decimals to 1, it would return 10.



## 2.6.1 Issues & Recommendations



<b>Issue #14</b>	<b>Decimals will be truncated when converting to lower decimals</b>
<b>Severity</b>	 LOW SEVERITY
<b>Description</b>	<p>Whenever an amount is converted to a lower decimal value, the decimals will be truncated.</p> <p>When converting 19999999999999999999 tokens to an amount with 6 decimals, the result will be 1999999 as the leftover decimals will be truncated, effectively decreasing the value.</p>
<b>Recommendation</b>	<p>Consider carefully inspecting all sections where this can happen, especially for future deployments on other chains where USDT is 6 decimals.</p> <p>Consider if this will then become an issue in practice.</p>
<b>Resolution</b>	 ACKNOWLEDGED
	<p>There is still truncation but in the scenario of normalizing to lower decimals, the mathematical calculation rounds up. During our assessment, we could not find an exploit which occurs due to rounding change, however, since a resolution round is a time-boxed review of the changes, we encourage the Ambit development team to execute further tests, ensuring that there can be no edge-cases due to rounding up for normalizing to lower decimals. The developer indicated that the rounding will be reversed in a newer commit. This will be validated once the commit has been provided.</p> <p>Resolution 2: The rounding change has been reversed.</p>

---

## 2.7 Libraries/PercentageMath

PercentageMath is responsible for calculating the percentage of an amount based on the provided basis points as well as the smaller value out of two values. It is used throughout the whole project.

### 2.7.1 Issues & Recommendations

<b>Issue #15</b>	<b>Typographical issues</b>
<b>Severity</b>	 INFORMATIONAL
<b>Description</b>	<u>L12</u> <pre>BPS public constant ONE_HUNDRED_BPS = BPS.wrap(10000);</pre> <p>This should be ONE_HUNDRED_PERCENT since 100 bps is 1%, making the current variable name a misnomer.</p>
<b>Recommendation</b>	Consider fixing the typographical issues.
<b>Resolution</b>	 RESOLVED





---

## 2.8 Libraries/PortfolioAssetExtensions

PortfolioAssetExtensions is used throughout the architecture with the functionality to fetch token addresses out of an array of PortfolioAsset structs. This token address is then used in most contracts to fetch the corresponding Asset struct.

### 2.8.1 Issues & Recommendations

No issues found.





---

## 2.9 Libraries/RayMath

RayMath is a customized version of AAVE's WadRayMath library (<https://github.com/aave/aave-v3-core/blob/master/contracts/protocol/libraries/math/WadRayMath.sol>), solely using the math calculations for WAD with implemented custom reverts and gas savings.



## 2.9.1 Issues & Recommendations

<b>Issue #16</b>	<b>add can overflow</b>
<b>Severity</b>	 LOW SEVERITY
<b>Description</b>	The add operation, unlike all other operations, does not have any overflow protection. It will therefore silently overflow which is inconsistent with the rest of the operations which do revert on overflow.
<b>Recommendation</b>	Consider simply using a high-level add which is protected against overflow.
<b>Resolution</b>	 RESOLVED This contract has been removed completely.



**Issue #17****Typographical issues****Severity** INFORMATIONAL**Description**L18

```
return Ray.wrap(1e27);
```

The literal 1e27 is already defined as the constant RAY, consider re-using it.

L44

```
if or(iszero(b), iszero(iszero(gt(a, div(sub(not(0), div(b, 2)), RAY)))))) {
```

iszero(iszero( does not appear to serve any purpose here. However, since as long as the optimizer is enabled, this gets optimized away, so it is not that big of a deal.

L52

```
/// @dev this peforms a POW operation using the algorithm defined here;
```

This comment has two typos. The correct spelling should be "performs" and "algorithm".

**Recommendation**

Consider fixing the typographical issues.

**Resolution** RESOLVED

This contract has been removed completely.

---

## 2.10 Libraries/USDMath

USDMath is responsible for safely converting origin values with 8 decimals to a value with target decimals.

### 2.10.1 Issues & Recommendations

<b>Issue #18</b>	<b>Typographical issues</b>
<b>Severity</b>	<span>INFORMATIONAL</span>
<b>Description</b>	<u>L7</u> <i>/// to make the code more expicit for when a function deals with a USD value.</i>  "expicit" should be spelled as "explicit".
<b>Recommendation</b>	Consider fixing the typographical issues.
<b>Resolution</b>	<span>RESOLVED</span>

---

## 2.11 Core/AddressRegistry

AddressRegistry is a simple registry contract which allows the admin to assign addresses to keys. It is essentially a global key-value database for the rest of the Ambit system. A key can only have one address assigned but different keys can link to the same address.

This contract is deployed externally and used as a global registry for the whole architecture. It is called by the AddressRegistryExtensions library using a valid hash.

### 2.11.1 Privileged Functions

- setAddress
- renounceRole [ role bearer ]
- grantRole
- revokeRole

### 2.11.2 Issues & Recommendations

No issues found.



---

## 2.12 Core/AssetStorage

AssetStorage is an externally deployed registry contract which keeps track of assets and their corresponding values such as:

- tokenAddress
- custodian
- priceOracle
- marketplaceAdapter
- maxSupply
- maxLTV
- liquidationDiscount



Any authorized address can add and change these values to/for any asset.

### 2.12.1 Privileged Functions

- setAsset



## 2.12.2 Issues & Recommendations

<b>Issue #19</b>	<b>Adjustment of LTV can make users vulnerable to liquidation</b>
<b>Severity</b>	 MEDIUM SEVERITY
<b>Description</b>	The setAsset function allows all parameters to be modified, including the LTV of an asset. Whenever the LTV is changed, this will immediately impact the borrowLimit of all users, eventually leading to unexpected liquidations.
<b>Recommendation</b>	Consider implementing a safeguard for this mechanism — i.e. a set and claim mechanism with a specific timestamp in between to prevent any accidental change.  It is also mandatory to communicate such a change with the community in a timely manner.
<b>Resolution</b>	 RESOLVED  The client has communicated they will be extremely careful with this. It can only be modified by the admins instead of authorized addresses.



**Issue #20**      **Lack of reasonable liquidationDiscount limit**

**Severity**      MEDIUM SEVERITY

**Description**      Lending protocols in general are always vulnerable to bad debt, which is the state where the borrower's debt position becomes larger than the collateral position, resulting in a disincentivization of paying back the loan.

To prevent this situation, positions can get liquidated once they become unhealthy, where not only the corresponding collateral but also a premium will be sent to the liquidator.

However, if the position is already in a very bad state, i.e. the borrowed position is 95% of the collateral and the premium is 6%, the position can never be fully liquidated anymore since there is not enough collateral to pay out the premium to the liquidator.

Moreover, a too large liquidationDiscount will make the position even more unhealthy.

**Recommendation**      Consider setting a reasonable upper limit for liquidationDiscount to receive a healthy state after liquidation as well as to keep liquidations valid as long as possible.

**Resolution**      ACKNOWLEDGED

**Issue #21**      **getAssets can run out of gas**

**Severity**      LOW SEVERITY

**Description**      The function fetches the Asset property from storage for every asset which was set. This can potentially consume a lot of gas if used by an external contract.

**Recommendation**      Consider implementing a reasonable upper limit for the \_count variable.

**Resolution**      ACKNOWLEDGED

**Issue #22****Gas optimizations****Severity** INFORMATIONAL**Description**L20 and 23

```
mapping (uint16 => address) private _indexes;  
uint16 private _count;
```

Using uint16 is redundant here and it might likely even increase gas costs. Consider moving to uint256. Note that the for-loop at line 118 must be adjusted as well.

---

It appears like the Asset struct is not optimally packed: The two BPS values can be moved under an address to save a full storage slot.

**Recommendation**

Consider implementing the gas optimizations mentioned above.

**Resolution** RESOLVED

**Issue #23****Typographical issues****Severity** INFORMATIONAL**Description**L19

```
// stores the indicies for the assets such that we can return  
all
```

This should say "indices" instead.

L29

```
modifier requireNonZeroAddress(address addr, string memory  
name) {
```

This modifier appears unused.

L63

```
_indexes[_count++] = asset.token;
```

It may make sense to also do a non-zero check on the token as to ensure that indices do not point to the default storage value which is all zeros.

L99

```
/// SLOAD that we can avoid by having the calling functions  
asset that they have the data they needed.
```

"asset" should be "assert" instead.

L100

```
function getAsset(address token, uint16 mask) public view  
returns (IAssetStorage.Asset memory) {
```

It should be noted that mask constants are presently uint8 which does not make much sense given that the mask is uint16. Consider marking all constants as uint16 instead.

**Recommendation**

Consider fixing the typographical issues.

**Resolution** RESOLVED

---

## 2.13 Governance/Overseer

Overseer is an emergency contract which allows the admin to shutdown the whole protocol by pausing the following contracts:



- DepositorVault
- Portfolio
- Market
- Liquidation
- Custodian

### 2.13.1 Privileged Functions

- shutdown



## 2.13.2 Issues & Recommendations

<b>Issue #24</b>	<b>shutdown functionality might run out of gas</b>
<b>Severity</b>	
<b>Description</b>	<p>Within the shutdown call, all registered assets are being fetched from the AssetStorage and then their corresponding custodian is paused.</p> <p>Under rare circumstances, if the amount of assets is very large, this can run out of gas, resulting in a DoS of the shutdown functionality.</p> <p>There are also other cases which might cause a pause call to revert — most notably when the Overseer does not have the ADMIN_ROLE for any of these contracts, and also if any of these contracts are already paused.</p>
<b>Recommendation</b>	<p>Consider limiting the amount of assets that can be registered to a reasonable upper value.</p> <p>More importantly, consider checking if all of the contracts are unpaused with an if-statement before pausing them. Next, consider adding a second helper function:</p> <pre>function shutdown(IPausable c) external onlyAdmin {     c.pause();     emit Paused(c); }</pre>
<b>Resolution</b>	 <p>Custodians will not be paused anymore, there implicit effects of not pausing custodians, such as other authorized addresses besides the Portfolio contract being able to withdraw, must be considered.</p> <p>We recommend pausing operations to be done extremely carefully and recommend carefully taking into account that custodians now need to be paused one by one, or by a new function that aggregates the pausing.</p>

**Issue #25**      **Pausing functionality might lead to undesired side-effects**

**Severity**      LOW SEVERITY

**Description**      Whenever the shutdown function is being called, supplying into the Portfolio, repaying borrowed funds and liquidations will also pause.

This can have the following risks:

- The occurrence of bad debt due to paused liquidation functionality
- Immediate liquidations after unpausing

**Recommendation**      Consider still allowing emergency liquidations as well as implementing a transition period where users can repay their debt or increase the collateral.

**Resolution**      RESOLVED

Users can still repay their debt within the Market contract.

We recommend that this is carefully considered as there might be instances where pausing this behavior is still desired. In this case, a proxy upgrade is currently necessary.

**Issue #26**      **Typographical issues**

**Severity**      INFORMATIONAL

**Description**      L14  
IAddressRegistry private immutable \_registry;  
  
\_registry should be made public to allow for users and reviewers to easily inspect it in the explorer.

\_\_\_\_\_  
  
shutdown should emit an event.

**Recommendation**      Consider fixing the typographical issues.

**Resolution**      RESOLVED

---

## 2.14 Governance/Treasury

Treasury is a simple token/ether storage contract with an ERC20 and ETH transfer functionality. It also includes a direct Ambit token burn functionality for Ambit tokens within the contract.

The treasury will receive funds during the following actions:



- During a liquidation mechanism, it will receive a share of the profits
- During a borrow action, it will receive a pro-rata share based on the total borrow amount
- During AmbitToken deployment, an initial supply of 100 000 000 tokens are being minted to the treasury
- During a contract sweep, the treasury will receive ERC20 tokens / Ether

### 2.14.1 Privileged Functions

- `transfer`
- `transferETH`
- `burnAmbit`



## 2.14.2 Issues & Recommendations

<b>Issue #27</b>	<b>Typographical issues</b>
<b>Severity</b>	 INFORMATIONAL
<b>Description</b>	<p><u>L21</u> IAddressRegistry private immutable _registry;  _registry should be made public to allow for users and reviewers to easily inspect it in the explorer.</p>
<b>Recommendation</b>	Consider fixing the typographical issues.
<b>Resolution</b>	 RESOLVED



---

## 2.15 Market/DynamicInterestRateModel

`DynamicInterestRateModel` is a simple helper contract responsible for calculating the current interest rate based on the `_baseRate`, current utilization ratio and multiplier.

A `_baseRate` of `1e27` represents 100% funding fee per year, while the multiplier reflects the percentage used as funding fee based on the utilization ratio. Therefore, a multiplier of `1e27` and a utilization ratio of 10% will represent an additional funding fee of 10% per year.



## 2.15.1 Issues & Recommendations

Issue #28

Interest rate mechanism is manipulatable due to utilization rate changes not always triggering an interest rate indexation

Severity



Description

The `DynamicInterestRateModel` bases its interest rate on the utilization of the depositor's vault supply (i.e. the percentage of this supply which is borrowed at any time). The interest rate increases together with the utilization rate.

However, this utilization rate is actually only fetched whenever users borrow or repay from the vault, not when users supply into or withdraw from the vault. This is an issue as the latter actions also change the utilization rate. This means that the new rate will be used retroactively whenever users supply or withdraw to the depositor vault.

A specific exploit scenario opens up because of this, for example:

1. Alice opens a huge borrow position on Ambit, and utilization rate is nearly 100%
2. Whenever users take actions that would index the borrow index with a very large number due to the high utilization rate rate, Alice frontruns them:
  - 2.1 Flashloan the full balance of a free flashloan USDT pool
  - 2.2 Deposit the USDT into the depositor vault
  - 2.3 Accrue the market liabilities (index the interest)
  - 2.4 Withdraw the flashloaned USDT
  - 2.5 Repay the flashloan

In this example, the actual utilization rate is 100% and interest should be huge. However, the interest gets indexed as if the utilization rate is only 10% for example. It should be noted that suppliers can also manipulate the interest rate upwards by flash withdrawing their deposits!

---

**Recommendation** Consider indexing the interest rate before all utilization rate changes: this includes deposits, withdrawals and borrow limit adjustments.

---

**Resolution**

 PARTIALLY RESOLVED

Protection for same block deposits/withdrawals have been added. Moreover, future changes will make the interest rate less sensitive to the state of the depositor vault.

The client has finally indicated they will monitor this carefully.

---

**Issue #29**

**Typographical issues**

**Severity**

 INFORMATIONAL

**Description**

L5 and 8

```
import { IERC20Metadata } from "../../dependencies/  
@openzeppelin/contracts/token/ERC20/extensions/  
IERC20Metadata.sol";  
import { Errors } from "../../libraries/Errors.sol";
```

These imports appear unused.

L19, 21 and 23

```
IAddressRegistry private immutable _registry;  
Ray private immutable _multiplier;  
Ray private immutable _baseRate;
```

These variables should be marked as `public` to allow them to be viewed from within the explorer.

---

**Recommendation** Consider fixing the typographical issues.

---

**Resolution**

 RESOLVED

---

---

## 2.16 Market/FixedInterestRateModel



`FixedInterestRateModel` is similar to the `DynamicInterestRateModel` — a helper contract which is called by the `Market` contract to return the current interest rate. Unlike the latter model, this contract uses a static interest rate which can be changed at any point by the owner and is not affected by the utilization ratio.



### 2.16.1 Privileged Functions

- `setRate`
- `transferOwnership`
- `renounceOwnership`



## 2.16.2 Issues & Recommendations

<b>Issue #30</b>	<b>Update of interest rate will change borrowIndex retroactively</b>
<b>Severity</b>	 MEDIUM SEVERITY
<b>Description</b>	<p>Whenever the setRate function is called, <code>_rate</code> will be changed while the market state is not updated beforehand.</p> <p>This will change the borrowIndex retroactively since the last time the market state was updated, resulting in potentially increased/decreased liabilities which can also lead to unexpected liquidations.</p>
<b>Recommendation</b>	Consider calling <code>accrueLiabilities</code> on the Market contract before the rate is being updated - this can be done programmatically within the code.
<b>Resolution</b>	 RESOLVED This variable cannot be changed anymore.

<b>Issue #31</b>	<b><code>_rate</code> can be set arbitrarily high</b>
<b>Severity</b>	 LOW SEVERITY
<b>Description</b>	Since the <code>_rate</code> of <code>1e27</code> represents 100% funding fees per year on the borrowing position, it might be desired to implement a reasonable upper limit on this variable.
<b>Recommendation</b>	Consider implementing a reasonable upper limit on the <code>setRate</code> function.
<b>Resolution</b>	 RESOLVED A check has been implemented, and the logic has been refactored such that 100% is <code>1e18</code> instead of <code>1e27</code> .

**Issue #32****Typographical issues****Severity** INFORMATIONAL**Description**L6 and 7

```
import { Ray, RayMath } from "../../libraries/RayMath.sol";  
import { IMarket } from "../../interfaces/market/  
IMarket.sol";
```

These imports appear unused.

L18

```
function setRate(Ray rate) external onlyOwner  
  
setRate should emit an event.
```

**Recommendation**

Consider fixing the typographical issues.

**Resolution** RESOLVED

---

## 2.17 Market/Liquidator

Liquidator is the entry point for every liquidation call. Any user can call the liquidation functionality to liquidate unhealthy accounts. Unlike traditional liquidation mechanisms, the Liquidator contract solely handles all funds necessary for a liquidation, thus it is not necessary for a user to provide funds to liquidate positions.

Once a position is successfully liquidated, the Liquidator will receive the corresponding collateral, which opens the path for three different scenarios:

- The liquidator receives the `baseAsset`, no further actions are necessary.
- The liquidator receives the `vaultToken` and immediately redeems it on the `DepositorVault` to receive the `baseAsset`.
- The liquidator receives an ERC20 token which is not the `baseAsset` and immediately attempts to sell it on a secondary UniswapV2 exchange for the `baseAsset`.

After the `baseAsset` is successfully received, the profit of the liquidation operation is calculated and distributed to the treasury, caller and donated to the vault, based on the configured fee.

In order to be able to liquidate positions, it is necessary for the Liquidator contract to have sufficient funds for the repayment of an unhealthy position.

### 2.17.1 Privileged Functions

- `sweep`
- `sweepETH`

## 2.17.2 Issues & Recommendations

**Issue #33**      **liquidateVaultToken is fundamentally flawed, preventing any amUSDT token liquidations from occurring**

**Severity**

 HIGH SEVERITY

**Description**

Within `liquidateVaultToken`, the depositor tokens can get liquidated if users supply them as collateral against which they borrow USDT. However, due to a fundamental issue within this function, the function will not work at all:

L97 (and 105)

```
uint amountReceived = vault.redeem(minShares, minAmountOut, msg.sender);
```

Within multiple sections of the function, the underlying redeemed USDT is sent to the user instead of kept within this contract, causing the contract to have insufficient USDT.

Furthermore, this function has excessive and misdirected rounding due to the reliance on the `exchangeRate` function.

**Recommendation**

Consider re-writing this function in a simplified manner. Ideally, do not rely on `exchangeRate()` at all since using this function results in potential rounding issues.

**Resolution**

 RESOLVED


The function has been simplified to the following steps:

- a) Redeem the liquidated amount without the premium
- b) Redeem the received premium

Additionally, the Liquidator is now the recipient of the tokens.



## Severity

 HIGH SEVERITY

## Description

Generally speaking, for lending protocols it is necessary to liquidate unhealthy positions as quickly as possible to prevent situations with bad-debt (a very prominent example is the BNB-Chain Venus liquidation). Usually, bots take care of the liquidations since they can act more quickly than humans.

Ambit Finance implements two liquidation mechanisms:

- a) The standard Liquidator liquidation
- b) The emergency liquidation via an authorized address that directly interacts with the MarketLiquidator

Whenever a position is liquidated, the collateral received is swapped to USDT — mostly this will be an ERC20 token as collateral.

SpotMarketMarketplaceAdapter is used for that purpose which essentially is an aggregator for UniswapV2 routers. However, since this adapter expects the slippage as parameter based on the `LiquidationDiscount`, the swap can revert due to two reasons:


- a) The overall liquidation amount is too large, resulting in a huge price impact.
- b) A very sophisticated attacker can frontrun the swap, manipulating the potentially used pools to differ more than 10% from the Chainlink price, resulting in a revert of the function and immediately backrunning the swap in an effort to not lose any funds due to MEV. Basically a sandwich attack of the liquidation functionality.

Both situations will result in a revert of the liquidation call, which can then result in bad debt.

## Recommendation

Consider simply using a DEX aggregator such as 1inch for that purpose.

## Resolution

 ACKNOWLEDGED

Ambit has indicated they will take this into account and manage their protocol accordingly.

## Severity

● LOW SEVERITY

## Description

Users can use the vaultToken they received for the USDT deposit within the DepositorVault as collateral within the Portfolio contract. If this position then becomes unhealthy, it will get liquidated which includes a redemption of the vaultToken. However, a malicious user can simply force the DepositorVault to a state where no USDT is left, circumventing the reserve logic as described within the *Reserve logic is redundant and can be circumvented* issue (Issue #69), causing a DoS of the liquidation call. (Or simply withdrawing staked USDT).

A user can therefore increase the borrow position using another token. This can result in bad-debt since the highest USD value will always be the vaultToken collateral, hence preventing the normal liquidation flow.

This issue is rated as low severity since the MarketLiquidation exposes an emergency liquidation function which allows the liquidation of the non-vaultToken position.


## Recommendation

Consider ensuring that a sufficient amount of USDT will be always within the vault, in an effort to cover such liquidation scenarios.

The scenario where a user a) deposits USDT with account 1 b) borrows USDT up to the reserve limit with account 2 and c) withdraws the USDT deposited in a) should also be prevented.

## Resolution

● ACKNOWLEDGED

**Issue #36****Profit of vaultToken liquidation can be reduced****Severity** LOW SEVERITY**Description**

During the redemption of the vaultToken, first the base amount is redeemed and then the premium:


```
// redeem the remaining amount which should be considered the profits
try vault.redeem(remaining, minAmountOut, msg.sender)
returns (uint received) {
    amountReceived += received;
} catch {
    // if for some reason the vault is empty we just keep hold of the remaining supply
}
```

A user with a valid deposit position / collateral position can simply frontrun this call, causing such a state in the DepositorVault that only the base amount can be redeemed and not the profit, effectively preventing any profit distribution.

**Recommendation**

Consider if this can become an issue, if yes, consider implementing a working reserve logic, which also applies to withdrawals within the DepositorVault contract.

**Resolution** ACKNOWLEDGED

**Issue #37****Allowing for unprofitable redemptions of vaultTokens is considered a risk****Severity** LOW SEVERITY**Description**

There are no requirements that a vault token liquidation should be profitable. This means that the Liquidator's USDT balance, which might be very sizable, can be used to cover the difference.

**Recommendation**

Consider if this can become an issue, if yes, consider implementing a **working** reserve logic which also applies to withdrawals within the DepositorVault contract.

**Resolution** RESOLVED

The client has indicated they ensure that this liquidation is never unprofitable. However, it is still not hardcoded in the code as a negative profit check which does not revert is still present:

```
uint256 profit = amountReceived < totalAmount ? 0...
```

We recommend reverting in this case unless there is a specific reason for it to remain this way.

In general, this issue will mostly re-introduce itself if logic is adjusted over time, allowing for negative redemption (e.g. redemption fees or loss bearing vaults). Also, the current code seems to indicate that negative profit is possible which is overall bad style. We therefore still recommend making the fix if there is no downside.

<b>Issue #38</b>	<b>Liquidation will not work if <code>_baseAsset</code> is not a USD stablecoin</b>
<b>Severity</b>	<span>● LOW SEVERITY</span>
<b>Description</b>	The slippage is calculated using the Chainlink price of the ERC20 token. However, if a different token than USDT is used as destination token, the slippage calculation will be flawed.
<b>Recommendation</b>	Consider this whenever a token other than USDT is used. We should also note that the whole audit was done under the assumption that USDT would exclusively be used as a base asset, and this audit should not be used to validate the trust for any system where this changes.
<b>Resolution</b>	<span>● ACKNOWLEDGED</span>

<b>Issue #39</b>	<b>Missing safeguard within <code>transferFees</code></b>
<b>Severity</b>	<span>● LOW SEVERITY</span>
<b>Description</b>	The <code>transferFees</code> function calculates the treasury and caller fee based on the profit. However, there is no check that this fee is in fact $\leq$ profit to ensure that the donation calculation does not underflow.
<b>Recommendation</b>	Consider implementing such a safeguard.
<b>Resolution</b>	<span>✓ RESOLVED</span> A check has been implemented to ensure that both fees cannot be larger than the received profit.

<b>Issue #40</b>	<b>Gas optimizations</b>
<b>Severity</b>	<span>INFORMATIONAL</span>
<b>Description</b>	<code>_registry</code> , <code>_treasuryFee</code> and <code>_callerFee</code> should be marked as <code>immutable</code> to save on gas.
<b>Recommendation</b>	Consider implementing the gas optimizations mentioned above.
<b>Resolution</b>	<span>RESOLVED</span>

<b>Issue #41</b>	<b>Typographical issues</b>
<b>Severity</b>	<span>INFORMATIONAL</span>
<b>Description</b>	<p><code>_registry</code>, <code>_treasuryFee</code> and <code>_callerFee</code> should be marked as <code>public</code> to allow for explorer inspection by users.</p> <p><code>IUniswapV2Router01</code>, <code>IMarketStorage</code>, <code>ICustodian</code>, <code>IPortfolioStorage</code>, <code>Normalizer</code>, <code>USDMath</code>, <code>AuthorizedAccessControl</code> and <code>MAX_SLIPPAGE_ALLOWED</code> appear unused. Consider removing them.</p>
<b>Recommendation</b>	Consider fixing these typographical issues.
<b>Resolution</b>	<span>PARTIALLY RESOLVED</span>



## Severity

 INFORMATIONAL

## Description

Whenever the `vaultToken` is redeemed, the current exchange rate of the `DepositorVault` contract is used to determine how much USDT should be received as a minimum.

However, this logic is useless since the `exchangeRate` is fetched within the same transaction where the `redeem` is done, thus it will not be changed before the `redeem` happens and stays the same.

Moreover, generally speaking, the `exchangeRate` can only be decreased under one special situation:

- No donation has been happened within the past six hours, and
- All shares have been redeemed

This situation will reset the `exchangeRate` to `1e18` and whenever this situation is present, the second `redeem` with the slippage calculation will always revert due to a lack of funds within the `DepositorVault`:

```
// redeem the remaining amount which should be considered the profits
try vault.redeem(remaining, minAmountOut, msg.sender)
returns (uint received) {
    amountReceived += received;
} catch {
    // if for some reason the vault is empty we just keep hold of the remaining supply
}
```

In fact, theoretically, if that edge-case where the `exchangeRate` is reset should have been covered, the `exchangeRate` should be redetermined after the first `redeem`.

Furthermore, allowing for unprofitable liquidations due to this slippage seems like a huge risk as unprofitable liquidations might reduce the USDT balance in the liquidator, which could very well lead to exploits to drain it! We extremely strongly urge the client to not allow for any loss-bearing liquidations because of this reason.

## Recommendation

Consider removing this unnecessary logic.

## Resolution

 RESOLVED

This has been removed.

---

## 2.18 Market/Market

Market is one of the main contracts within the Ambit Finance ecosystem and allows users to borrow and repay the underlying asset of the DepositorVault. Users can borrow against their collateral positions from the Portfolio contract, up to a specific percentage based on the maxLTV for the different collateral configurations. Over time, users will pay interest on their borrowed position which can be either a dynamic rate or a static rate based on the configuration of the contract.

During repayment, a distinction is made between the actual debt position and the accrued interest on the position. While the debt position is repaid as usual, the interest is donated to the DepositorVault contract, essentially generating a linear yield for the staking positions inside the DepositorVault.


Moreover, whenever a user borrows tokens, a `_borrowingFee` is applied, which is then subtracted from the to-be received tokens but still applied as liability, therefore, users will immediately have a higher debt position than they have tokens received. This fee is then sent to the treasury contract.

### 2.18.1 Privileged Functions

- `sweep`
- `sweepETH`
- `pause`
- `resume`
- `setBorrowingFee`
- `borrow (broker)`
- `repay (broker)`
- `repayOutstandingLiabilities (broker)`



## 2.18.2 Issues & Recommendations

<b>Issue #43</b>	<b>Stablecoin depositor vault can be fully drained if an asset with 20 decimals or more is ever added as collateral due to an overflow vulnerability in the borrowing limit calculation</b>
<b>Severity</b>	 HIGH SEVERITY
<b>Location</b>	<u>L132</u> <pre>uint total = amount.mulDiv(USD.unwrap(price), uint64(10 ** decimals));</pre>
<b>Description</b>	<p>The Ambit infrastructure allows for users to supply various assets into the system as collateral to borrow against. The <code>getBorrowLimit</code> functions then sum up all the dollar values of these assets then multiply them by their LTV (percentage of their value usable as borrowing power). This represents the borrowing limit.</p> <p>However, due to an unsafe cast at line 132, if the underlying supplied asset has a decimal number of 20 or more, an overflow occurs. In case this overflow occurs, the borrowing limit can be massively inflated. This means that as soon as an asset is added with 20 or more decimals, anyone using that asset as collateral is likely to get many multiples of its value in borrowing power.</p> <p>Exploit scenario:</p> <ul style="list-style-type: none"><li>- The team adds "PRECISEUSD" as a collateral, a token with 20 decimals.</li><li>- Due to the overflow, <code>uint256(10**20)</code> overflows to <code>7766279631452241920</code>. The correct value would have been <code>100000000000000000000</code>, about 12 times larger than the overflow value.</li><li>- The exploiter supplies \$1m PRECISEUSD. The total calculation becomes <math>(1 \text{ million} * 1e20) * \text{price} / 7766279631452241920</math> or approximately <math>1 \text{ million} * 12 * \text{price}</math>. This means that the user gets 12 times more borrowing power and can fully drain the depositor vault's underlying stablecoins. If a 75% LTV is used, up to \$9m underlying stablecoins can be drained for a supplied value of \$1m.</li></ul>

---

This issue therefore results in a full drain if such a high-decimal asset is ever added. There are quite a few assets with 20 or more decimals on-chain, making this issue realistically possible. However, to our knowledge, none of the major on-chain assets have 20 or more decimals, so it is also possible that this will never occur.

---

**Recommendation** Consider removing the uint64 cast fully and simply retaining a full 256 bit variable.

---

**Resolution** 

---

## Issue #44 ensureHealthyAccount returns true for a score of 100

**Severity** 

---

**Description** ensureHealthyAccount is called by various external contracts and represents if an account is still healthy. For example, during a withdrawal in the Portfolio contract, this function is called. However, it will also return true if threshold = 100, essentially returning a flawed value.

Portfolio code:

```
function ensureHealthyAccount(address account) private view
{
    IMarket market = _registry.getMarket();
    market.ensureHealthyAccount(account, 100);
}
```

Market code:

```
if (borrowLimit * 100 / liabilities >= threshold) {
```

---


**Recommendation** Consider changing `>=` to `>`. It might also make sense to carry over the liabilities division to the other side (as a multiplication) to significantly increase precision.

---

**Resolution** 

While the 100 threshold is still unchanged, an address is now only considered as unhealthy when reaching a threshold of 99. This fix was implemented within the MarketLiquidation contract.

---

**Issue #45****Borrowing can result in liquidation****Severity** LOW SEVERITY**Description**

Within `borrowInternal`, the following check is done to ensure that an account cannot become unhealthy by a borrow action:

```
if (userLiability.liabilities + amount > borrowLimit) {  
    revert Errors.Market_BorrowLimitExceeded({  
        borrowLimit: borrowLimit,  
        liabilities: userLiability.liabilities + amount  
    });  
}
```

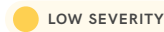
However, this will not revert if the values are equal, hence a user can get immediately liquidated.

**Recommendation**

Consider reverting for `>=` as well.

**Resolution** RESOLVED

An address is now still considered healthy if both values are equal, hence a revert for this scenario is not necessary anymore.

**Severity****Description**

Interest is only distributed to the vault at the very end of the borrow period right when the underlying funds are returned. This can cause significant misalignment in incentives as when a large sum of underlying currency is returned to the vault, depositing APR will suddenly become much higher. In this case, it should in fact be lower as the vault now has significant excess liquidity and does not need more.

Furthermore, given that interest is only paid at the end of the borrow period, suppliers might not have much incentive to continue supplying if most borrowers become long-term borrowers. They might therefore withdraw their funds and cause the utilization rate to constantly fluctuate. The end result is that utilization will periodically become very high and then very low, due to the combination of these misalignments.

This can even result in a small exploit to avoid paying a lot of interest. The exploit could for example enable someone to only pay a fraction of the interest by being the largest depositor in the vault for a week (or whatever the distribution period is), right before the protocol pauses preventing anyone else from joining in to claim the yield. Additionally, the capital that actually provided the funds for that large and long borrow is never directly repaid by that interest. Instead, new entrants will likely get most of the yield.

Exploit scenario:

- 1) Exploiter Alice creates a long and does not repay it for 10 years. She provided enough collateral to not be liquidated over this timespan.
- 2) After those approximate 10 years, the team has to do maintenance and pauses the protocol for a week.
- 3) Alice notices this pausing in the mempool and frontruns the transaction by finally repaying her loan which kicks in the interest distribution. She furthermore stakes a large amount of underlying stablecoins into the depositor vault to become a 90% owner of the depositor vault.

- 
- 4) As the protocol is paused, users cannot deposit in the vault to capture the huge amount of interest she just donated to the vault by repaying her loan. Instead, she is able to capture it almost entirely herself as she makes up 90% of the vault.
- 

**Recommendation** Consider whether there are ways to collect the interest to the borrowIndex over time. The challenge with such a design is that the borrowIndex would not be fully represented by liquid assets in this case, as some assets would still need to be returned.


A slightly less elegant solution could be to force users to repay after a certain period.

---

**Resolution**

ACKNOWLEDGED



**Issue #47****Automatic reduction of the repay amount to outstanding liabilities could cause issues for integrating applications who are unaware of this****Severity** LOW SEVERITY**Location**L207

```
amount = Math.min(amount, userLiability.liabilities);
```

**Description**

If an amount higher than the user's liability is repaid, this will result in the function simply reducing the amount to the total outstanding liability.


Though this is great, not all integrations by third-party smart contracts might realize that this happens and can lead to unexpected side-effects for these applications.

An example could be a third-party application who pulls in amount, and then assumes that the full amount would be forwarded to the market when they call repay.

**Recommendation**

Consider whether it makes sense to only do this reduction in case the amount is equal to the maximum integer, and otherwise revert if the amount is larger than the liabilities. Next, consider preventing the max integer from being supplied in the normal repay functions, to ensure that they are not accidentally mis-used.

**Resolution** ACKNOWLEDGED

**Issue #48****Governance risk: `_borrowingFee` lacks upper limit****Severity** LOW SEVERITY**Description**

This struct value can be set without any limitations, which can result in a loss of user funds as it means the fee to open borrowing positions can be set to a large value of up to 100% of the borrowed amount and should therefore implement a reasonable limit for the BPs.

Governance exploit example:

- User A submits a transaction to borrow \$1,000.
- Governance keys have been leaked, and the hacker sets the borrow fee to 100%.
- User A pays his whole borrowed amount to the hacker, and receives no stablecoins themselves.

**Recommendation**

Consider implementing a safeguard so the BPs cannot be set to a value larger than a certain reasonable threshold.

**Resolution** RESOLVED

## Severity

 INFORMATIONAL

## Description

The market state is calculated under every circumstance, even if the `lastUpdate` was at the current timestamp:

```
function calculateMarketState(IMarketStorage.MarketState
memory marketState) internal view returns
(IMarketStorage.MarketState memory) {

    uint256 ellapsed = block.timestamp -
marketState.lastUpdate;

    if (marketState.lastUpdate > block.timestamp) {
        revert Errors.Market_InvalidMarketState({
            blockTimestamp: block.timestamp,
            lastUpdate: marketState.lastUpdate
        });
    }

    Ray rate =
_registry.getMarketInterestRateModel().calculateRate();

    return calculateMarketState(marketState, rate,
ellapsed);
}
```

This will unnecessarily waste gas since the market state is almost updated during every transaction within the Market contract, potentially updating it multiple times per block.

## Recommendation

Consider simply returning early if no time has elapsed since the last update.

## Resolution

 RESOLVED

A check has been implemented within the `accrueLiabilities` function.



## Severity

INFORMATIONAL

## Description

Line 52

```
IAddressRegistry private _registry;
```

This variable should be marked as `immutable` (and `public`).

Line 128

```
uint8 decimals = custodian.getUnderlyingAsset().decimals();
```

Calling this sequence seems rather wasteful as this value is not supposed to change. However, we understand that this can be left as-is for code simplicity purposes. The solution would probably be to encode the decimals into an immutable variable within the custodian, and return these as well. This would of course make the code less clean.

Line 147

```
function borrowInternal(address account, uint256 amount,  
address receiver) private whenNotPaused  
requireNonZeroAmount(amount) {
```

The modifiers appear redundant here as they appear present on the external functions already.

Line 231

```
underlyingAsset.safeIncreaseAllowance(address(vault), debt +  
interest);
```

`amount` seems to be identical to `debt + interest` and could therefore be cheaper to use as the approval value.

Lines 306-311

```
if (marketState.lastUpdate > block.timestamp) {  
    [...]  
}
```

This code appears unreachable and can probably be removed.

## Recommendation

Consider implementing the gas optimizations mentioned above.

## Resolution

PARTIALLY RESOLVED

**Description**

Lines 4, 20, 21, 27 and 39

```
import { Address } from "../../dependencies/@openzeppelin/contracts/utils/Address.sol";
import { IInterestRateModel } from "../../interfaces/market/IInterestRateModel.sol";
import { IPortfolio } from "../../interfaces/portfolio/IPortfolio.sol";
import { Ray, RayMath } from "../../libraries/RayMath.sol";
using Address for address payable;
```

These imports (not Ray) are unused and can therefore be removed.

Line 52

```
IAddressRegistry private _registry;
```

This variable should be marked as public (and immutable).

Line 63

```
receive() external payable { }
```

This line appears to be unused and might introduce accidental ETH sending which cannot be taken out of this contract except for with the Sweepable dependency.

Lines 245-248

```
uint percentage = amount.mulDiv(10 ** decimals,
userLiability.liabilities);
```

```
interest = userLiability.liabilities -
userLiability.borrowed;
interest = interest.mulDiv(percentage, 10 ** decimals);
```

interest is calculated via division-before-multiplication, resulting in less than optimal rounding behavior. We strongly urge the client to re-write this code into a simpler structure which does not unnecessarily round:

```
interest = amount.mul(userLiability.liabilities -
userLiability.borrowed).div(userLiability.liabilities)
```

---

`accrueLiabilities` lacks a `whenNotPaused` modifier, allowing it to still be called in a paused state, which may not be desired.

---

It should also be noted that we recommend adding reentrancy guards on all external functions in case the team wishes to redeploy on a chain where the underlying stablecoin of the depositor vault has, or could eventually have through upgrades, reentrancy hooks.

---

`calculateMarketState` alongside other code locations repeatedly has "elapsed" mis-spelled as "ellapsed".

---

The contract exposes two `repayOutstandingLiabilities` functions of which one is only for authorized brokers. However, within the whole architecture this function is never used. Moreover, it could be abused to do dusting attacks in an effort to front-run full repayments.

---

It should be noted that the `getBorrowLimit` functions are rather inconsistent in their return values as some return USD-based decimals (8) while others return a value in the decimals of the depositor vault's underlying value.

---

**Recommendation** Consider fixing the typographical issues.

---

**Resolution**

 PARTIALLY RESOLVED

No additional reentrancy guards have been added to this contract.

---

---

## 2.19 Market/MarketLiquidation

MarketLiquidation handles the liquidation logic for borrowing positions within the Ambit Finance ecosystem. The `liquidate` function is triggered by the `Liquidator` contract, repays a specific amount of an unhealthy account's borrowing position, and takes the collateral in exchange.

A position is considered unhealthy as soon as the liabilities reach the `borrowLimit`, therefore users can increase their health factor by repaying liabilities or increasing their collateral.

The contract allows for two liquidation possibilities:

- Liquidate a borrowed position and receive a specific desired collateral asset.
- Liquidate a borrowed position and receive the collateral with the largest position.

The liquidator therefore repays the debt and receives the full corresponding collateral, including the premium. Collateral is withdrawn from the corresponding Custodian contract and the account's shares are decreased. The latter methodology is used by the `Liquidator` contract, we highly assume that the first methodology is reserved for some emergency liquidation scenarios.

It is only possible to liquidate a maximum of 50% of the total debt.

We reiterate that within this contract, positions are seen as liquidatable once they have a 100% health factor — this may be unexpected for users and lead to issues in other parts of the system if they assume 100% is still healthy.

It should be noted that as with several other sections of the codebase, the liquidation logic does not strictly adhere to checks-effects-interactions, a pattern we strongly recommend for complex codebases like this. We have emphasised multiple times that the team should strictly refrain from adding any components

that introduce reentrancy vectors such as reentrancy tokens or swap routes as this would break the key security properties of the system.

## 2.19.1 Privileged Functions


- `liquidate`



## 2.19.2 Issues & Recommendations

<b>Issue #52</b>	<b>findLargestPosition does not include collateralFactor</b>
<b>Severity</b>	<span>● LOW SEVERITY</span>
<b>Description</b>	This function returns the largest position as USD value. However, since tokens can have a different collateral ratio/factor, this could return even positions without a collateral factor of zero, indicating that these positions are never used for borrowing purposes.
<b>Recommendation</b>	Consider not checking for the largest USD-Value per se but also including the collateral factor. Note that liquidating the positions with a lower collateral factor might be even more desirable as it is more effective at turning an account solvent again.
<b>Resolution</b>	<span>● ACKNOWLEDGED</span> The client plans to implement specific features to disable tokens as collateral at a later point in time.



**Issue #53****calculateLiquidation potentially compares values in different denominations****Severity** LOW SEVERITY**Description**

calculateLiquidation includes the following line:

```
maxAmount = Math.min(maxAmount,  
totalSupply.mulDiv(discountedPrice, scalar,  
Math.Rounding.Down));
```

This will not work if maxAmount is denominated in a different value than USD, or if discountedPrice uses a USD Chainlink oracle for calculation purposes.

This issue has been downgraded from HIGH to LOW since the client makes an explicit assumption that they would never put any other asset than a stablecoin as the underlying asset of the depositor vault.


It is left in as a reminder that this audit specifically assumes that USDT will always be used as the underlying asset and that this audit does not apply for deployments with a different underlying borrowable token than a stablecoin.

**Recommendation**

Consider using a different oracle whenever a baseAsset other than USDT is being used.

**Resolution** RESOLVED

Ambit Finance was developed to only work with USD oracles.

**Issue #54****Funds are withdrawn directly from the Custodian instead of the PortfolioBroker****Severity** LOW SEVERITY**Description**

After the repayment is done while liquidating, the total supply is withdrawn from the custodian to the liquidator. This is not the pattern that Ambit follows on its architecture because it should be calling the portfolio withdraw function instead of the custodians withdraw.

During the liquidation, the supply is withdrawn from the Custodian instead of the Liquidator. This breaks the desired function flow and might result in edge-cases.


**Recommendation**

Consider withdrawing from Portfolio instead Custodian.

We note that this might have been done to circumvent health checks. Keep this in mind as being able to liquidate such positions might be a business decision/trade-off.

**Resolution** RESOLVED



**Issue #55****requireUnhealthyAccount passes if liabilities are zero****Severity** LOW SEVERITY**Description**

When an account has zero liabilities, this account is considered as healthy. However, this modifier does not revert for the edge-case that liabilities are zero:

```
modifier requireUnhealthyAccount(address account) {
    IMarket market = _registry.getMarket();

    uint256 liabilities = market.getLiabilities(account);

    if (liabilities > 0) {
        uint borrowLimit = market.getBorrowLimit(account);

        uint health = borrowLimit * 100 / liabilities;

        if (health >
Constants.UNHEALTHY_HEALTH_SCORE_THRESHOLD) {
            revert
Errors.MarketLiquidation_AccountTooHealthy(account, health);
        }
    }


    -;
}
```

This issue was only rated as low severity since this flaw cannot be abused.

**Recommendation**

Consider reverting for this edge-case as well.

**Resolution** RESOLVED

**Issue #56****Highest position is potentially truncated****Severity** LOW SEVERITY**Description**

Whenever the the positions are fetched, they are denominated as USD value with 8 decimals:

```
(, USD[] memory totals) =  
portfolio.getPortfolioValue(portfolioAssets);
```

Once the logic has determined which position in the array has the largest USD value, it is normalized to the `_baseAsset` denomination:

```
uint total =  
totals[j].normalize(underlyingAsset.decimals());
```


In Ambit Finance's deployment scenario, this will be USDT with 18 decimals on BSC. However, that is not a guaranteed factor, since it could also be a token with 6 decimals, that would essentially result in the last 2 decimals being truncated, returning a smaller position size than desired.

This will then decrease the liquidation value.

**Recommendation**

A fix is non-trivial, since truncating cannot be prevented. Do note that for complex fixes, there may be an extra fee levied to validate the code.

**Resolution** ACKNOWLEDGED

**Issue #57****Potential mismatch between maxAmount and liquidationSupply****Severity** LOW SEVERITY**Description**

Within calculateLiquidation, two amounts are calculated:

- a) Amount which is necessary to liquidate a position
- b) Corresponding collateral value which is received for this liquidation

Since both values are corresponding, they will perfectly match, however, the following line might decrease a), therefore leading in a small additional loss for the liquidated user:

```
maxAmount = maxAmount.normalize(decimals,  
underlyingAsset.decimals());
```

The normalize function will truncate maxAmount whenever the underlying asset has fewer decimals.


**Recommendation**

Consider if this loss is acceptable, if not, the whole logic needs to be refactored.

Do note that for complex fixes, there may be an extra fee levied to validate the code.

**Resolution** ACKNOWLEDGED

## Severity

 LOW SEVERITY

## Description

Within `calculateDecimals`, the last step of calculating the `liquidationSupply` is the comparison of the `totalSupply` with the `liquidationSupply`:

```
liquidationSupply = Math.min(totalSupply,  
liquidationSupply);
```

However, the comparison does not use the same decimal denominations as `totalSupply` is still the normalized value, whereas `liquidationSupply` has already been converted to the correct value:

```
(decimals, maxAmount, discountedPrice, totalSupply) =  
normalizeAmounts(asset, maxAmount, discountedPrice,  
totalSupply);
```

```
liquidationSupply = liquidationSupply.normalize(decimals,  
ICustodian(asset.custodian).getUnderlyingAsset().decimals())  
;
```

Fortunately, this will not result in any issues since the `totalSupply` will always be larger than the `liquidationSupply`, hence it can never impact the return value.


## Recommendation

Consider comparing the values in the same denominations.

## Resolution

 RESOLVED

This has been removed completely.

**Issue #58****Small positions can never be fully liquidated****Severity** LOW SEVERITY**Description**

The contract only permits the liquidation of up to 50% of a position. This means that after some time, it is simply not worth it to further liquidate the position and the protocol might accumulate bad debt because of it.


**Recommendation**

This is a fundamental issue as a user can always open many small positions which would cost too much gas to liquidate. A solution could be to add a minimum position size (which must be fully withdrawn if a withdrawal would go below it), but we understand that this may be too messy to implement.

Consider also carefully investigating whether this bad debt can be managed or not.

**Resolution** RESOLVED

A mechanism which refunds the value to a user in the scenario where the liquidation is initiated with a larger value than the liabilities has been introduced. However, this change introduces a higher penalty towards users which can be denominated by the premium based on  $\min(\text{threshold}, \text{positionSize})$ , depending on whether the forwarded balance within the Liquidator is sufficient to cover the value. Therefore, if a `positionSize` is large and the forwarded value from the liquidator is large, the user can experience a way larger penalty which is now based on the `smallAccountThreshold` instead of the repaid amount. The admin should therefore carefully select the `smallAccountThreshold` value.

**Issue #59****Liquidation functions lack a minimum received slippage check for manual liquidations****Severity** LOW SEVERITY**Description**

If the Ambit team ever wants to manually liquidate a position, they would have to add a slippage parameter to the `liquidate` function to ensure they pay and receive values within reason.

This could particularly be a problem if a manual liquidation is frontrun.

**Recommendation**

Consider whether it makes sense to add validation parameters to the `liquidate` functions which ensure that the liquidation went as expected. This is particularly useful for the secondary `liquidate` function which is not used by the automatic liquidator and therefore presumably used solely by manual liquidation — perhaps only adding checks to this function (in the form of a `minDiscountedPrice` or something) is most sensible.

**Resolution** ACKNOWLEDGED**Issue #60****Typographical issues****Severity** INFORMATIONAL**Description**

`SafeERC20`, `ASSET_FIELD_MARKETPLACE_ADAPTER`, `ASSET_FIELD_MAX_SUPPLY`, `ASSET_FIELD_MAX_LTV`, `IMarketBroker`, `IMarketStorage`, `IDepositorVault` and the `requireNonZeroAmount` modifier are unused and can therefore be removed.

\_\_\_\_\_

`_registry` should be marked as public to allow users to inspect it via the explorer.

**Recommendation**

Consider fixing these typographical issues.

**Resolution** PARTIALLY RESOLVED

---

## 2.20 Market/MarketStorage



MarketStorage contains the sensitive storage of the Market contract, such as user liabilities and the current market state.

It is called whenever the market state is updated and whenever users execute a borrow or repay transaction on the Market contract.

### 2.20.1 Privileged Functions

- `setMarketState`
- `setUserLiabilities`

### 2.20.2 Issues & Recommendations

<b>Issue #61</b>	<b>Typographical issues</b>
<b>Severity</b>	 INFORMATIONAL
<b>Description</b>	<p>The parameters for the <code>setMarketState</code> and <code>setUserLiabilities</code> function can be marked as <code>calldata</code>.</p> <p>---</p> <p>The setters lack events, though this might be valid as it may be considered wasteful as events might be generated appropriately within the Market contract already.</p>
<b>Recommendation</b>	Consider fixing these typographical issues.
<b>Resolution</b>	 RESOLVED

---



## 2.21 Marketplace/ DepositorVaultMarketplaceAdapter

DepositorVaultMarketplaceAdapter is a simple marketplace adapter which is responsible for redeeming the vaultToken in the DepositorVault / deposit the baseAsset in the DepositorVault contract. The result of the exchange transaction is always based on the current vault exchange rate, which is based on the yield distribution within the vault to date.





## 2.21.1 Issues & Recommendations

<b>Issue #62</b>	<b>normalize can truncate price if vaultToken has less than 8 decimals</b>
<b>Severity</b>	 LOW SEVERITY
<b>Description</b>	<p>The minimum amount to receive for redemption is determined as follows:</p> <pre>uint minAmountOut = amount.mulDiv(price.normalize(decimals), 10 ** decimals);</pre> <p>However, if the vaultToken has fewer than 8 decimals, this will result in a truncated price, potentially decreasing minAmountOut.</p> <p>The same issue applies to the buy function where minSharesOut is increased which can potentially DoS such a function call.</p>
<b>Recommendation</b>	Consider normalizing using the correct decimals instead to convert from underlying to share decimals.
<b>Resolution</b>	 RESOLVED The logic has been refactored and truncation is prevented.

**Issue #63****Typographical issues****Severity** INFORMATIONAL**Description**Line 28

```
IAddressRegistry private _registry;
```

This variable can be marked as public to make explorer inspection easier for users and can be marked as immutable to save on gas.

---

We do not understand the benefit of using `safeIncreaseAllowance`. `forceApprove` recently introduced in OpenZeppelin makes much more sense in our opinion.

**Recommendation**

Consider fixing the typographical issues.

**Resolution** RESOLVED

---

## 2.22 Marketplace/Marketplace

Marketplace has two main functionalities for users:

- Allowing users to sell collateral in their portfolio to the `baseAsset` and repay their debt.
- Allowing users to leverage their existing collateral position, similar to the traditional looping mechanism by taking a flashloan from the `DepositorVault` contract, purchasing a valid collateral token, supplying it as collateral and repaying the flashloan and fee with the remaining borrowable amount. Moreover, instead of swapping the flashloan to a valid ERC20 token, a user can also deposit it into the `DepositorVault`, receiving the `VaultToken`.

## 2.22.1 Issues & Recommendations

**Issue #64**      **The brokerage functions of the marketplace contain faulty internal states, allowing for any reentrant behavior to potentially exploit the user and force liquidation**

**Severity**

 HIGH SEVERITY

**Description**

The Marketplace is what is considered a broker within the Market system. A broker is allowed to make a set of transactions for a user with the intermediary states for that user being “insolvency”.

This is exclusively safe if there is absolutely no reentrancy concern at any point during these calls. Unfortunately, this is not necessarily the case for Marketplace — within Marketplace, there are multiple opportunities for eventual reentrancy during these transactions: e.g. if an upgradeable DEX is added which upgrades to a malicious implementation, or if 1inch or similar is ever supported where an off-chain or on-chain system can configure an arbitrary swap route.

If such a swap route can be maliciously configured by an exploiter, it can be configured to attempt to liquidate the user who is making a swap. Since during these swap transactions the user account is insolvent, they will in fact be liquidatable and the exploiter will be able to collect the liquidation bonus.

Alongside this example exploit, the fact that intermediary faulty states exist within these transactions can be a concern for exploits that benefit from “read-only-reentrancy” and the likes. It is therefore strongly recommended that no such faulty intermediary states exist.

---

**Recommendation** Consider giving up the idea of these brokers being permitted to make accounts temporarily insolvent as it makes the contracts less secure. Instead, it might be (though less performant) safer to simply flashloan the amount of USDT that would minimally need to be sold beforehand.

Alternative designs do exist where a temporary “insolvency” can be safely done. But such designs need to be natively integrated into the core architecture to be safe and need to be extremely carefully considered.

Consider whether it makes sense to have a global reentrancy guard for any action which comes from an unprivileged account into the system.

---

## Resolution



The contract has been refactored and a new approach has been used for that logic. We recommended the client to explicitly include safeguards within the core but this is not done. Instead, each authorized account still needs to carefully check that the user is solvent at the end of its operations, and ensure that no external calls or exception logic gets made in any insolvent state.

This is less than ideal in our opinion, and we have reminded the client that we would prefer the core to make this solvency guarantee.

---

**Issue #65****borrowFeeAmount is not accounted into slippage****Severity** MEDIUM SEVERITY**Description**

The user can determine a certain price and slippage when executing a leverager transaction.

For example, if a user takes out a flashloan of 500 USDT and determines a price of 1 (100000000) and a slippage of 100 BPs, this means that the user expects at least 495 (let's say USDC) as output amount.

However, this calculation does not account for the borrowFee, which is deducted from the input amount for the swap:

```
uint borrowFeeAmount =
market.getBorrowingFee().calculate(context.amount +
context.fees);

uint availableAmount = context.amount - borrowFeeAmount;

uint amountOut =
IMarketplaceAdapter(context.marketplaceAdapter).buy(
    availableAmount,
    context.price,
    context.slippage);
```

Users will therefore always accept more slippage than desired.

**Recommendation**

Consider implementing logic that accounts for the borrowFee additionally.

**Resolution** RESOLVED

Logic has been implemented which adjusts the slippage accordingly.

**Issue #66****Healthy state is not guaranteed****Severity** MEDIUM SEVERITY**Description**

Within the MarketplaceVendor contract, the estimated USDT value received for the sale of the specific collateral amount is calculated via `estimateSellAmount`.

In certain scenarios, if the provided slippage is unreasonably large and the user is already on the verge of liquidation, this USDT value might be insufficient to maintain a healthy ratio whenever the corresponding collateral value is withdrawn.

While the function at the end ensures that the position must be healthy, it certainly reverts for that situation.

However since the swap is executed after the withdrawal, a reentrancy-attack to liquidate the user can still happen here.

This issue is only rated as medium since the likelihood of this happening is very low and the user would need to select a very large slippage for this operation.


**Recommendation**

Consider adding a health check directly after the collateral withdrawal.

**Resolution** RESOLVED

This check has been implemented post-withdrawal as well.



**Issue #67****available calculation does not work for baseAssets that are not USDT****Severity** LOW SEVERITY**Description**

The following line will return an incorrect value whenever a baseAsset other than USDT is used:

```
uint available = market.getBorrowLimit(msg.sender) -  
market.getLiabilities(msg.sender);
```

This potentially results in a revert of the whole function due to an underflow.

**Recommendation**

Consider either implementing a different logic or simply ensuring that only USDT / a coin, with the same denomination as the liabilities is used.

Note that this will be resolved on the note of this not being a concern since we believe these functions are all specifically designed for USDT buying and selling.


**Resolution** RESOLVED

No other assets will be used.





## Severity

 LOW SEVERITY

## Description

The following line estimates the value which can be used for leveraging the portfolio, depending on the current free collateral and the LTV:

```
uint maxAmount = estimateMaxAmount(available, asset.maxLTV);
```

As an example, if a user has 100 USDT of free collateral and the asset which is desired to leverage as collateral has a LTV of 80%, this will result in a flashloan amount of 500. Since  $100$  and  $500 * 0.8 = 400$ , which is the amount that can be borrowed again to repay the flashloan.

The issue with this calculation is that it returns the value under the circumstances that this will be the value supplied as collateral again. However, that is incorrect because

- a) the flashloan fee,
- b) the swap potentially encounters slippage, and
- c) the borrow fee


will be deducted during the overall process, hence making this calculation useless.

*\*This issue was only rated as low severity since a health check is enforced at the end of the function.*

## Recommendation

Consider including these factors in that check.

## Resolution

 ACKNOWLEDGED

**Description**

Whenever a user executes a leveraged buy call, the user can choose their desired amount as flashloan. However, if the vault has insufficient tokens, the flashloan request will only be executed using the funds left in the vault:

```
maxAmount = Math.min(amount,  
vault.maxFlashLoan(underlyingAsset));
```

Afterwards, the context is being encoded, which is including the initial provided amount:

```
bytes memory context = abi.encode(msg.sender, token,  
asset.marketplaceAdapter, amount, price, slippage);
```

This context is then encoded within the callback function of the flashloan:

```
FlashLoanContext memory context;  
context.amount = amount;  
context.fees = fee;  
  
(  
    context.account,  
    context.asset,  
    context.marketplaceAdapter,  
    context.maxAmount,  
    context.price,  
    context.slippage  
) = abi.decode(data, (address, address, address, uint, USD,  
BPS));
```

Afterwards, the following check is executed:

```
if (context.amount < context.maxAmount) {  
    // dont continue with the swap if we didn't get all the  
funds we were after  
    return CALLBACK_SUCCESS;  
}
```

---

Which essentially compares the initial value we provided and the received flashloan amount.

In the scenario where the vault has insufficient tokens, as mentioned above, this will simply return the CALLBACK\_SUCESS string. However, since no approval has been granted and no fees have been accumulated, this will always revert.

---

**Recommendation** Consider simply reverting directly in that scenario.

---

**Resolution** 

---

**Issue #70** **sell does not return the total amountOut; instead, it returns the amountOut after the liabilities repaid have been deducted**

**Severity** 


---

**Description** The sell function appears to intend to return the amount of USDT that the asset was sold for. However, it appears to instead return the sold amount after the liability repayment has been deducted, which may be erroneous.

---

**Recommendation** Consider whether this is the desired amount to return or whether the total amount received should be returned instead.

---

**Resolution**   
The return value has been removed completely.

---

**Issue #71**      **Missing check that marketplaceAdapter is set when buying from your portfolio**

**Severity**      INFORMATIONAL

**Description**      When leveraging from the portfolio, the sell function checks that a marketplaceAdapter is actually set:

```
if (asset.marketplaceAdapter == address(0)) {  
    if (asset.token != address(baseAsset)) {  
        revert  
        Errors.Marketplace_MissingMarketplaceAdapter(asset.token);  
    }  
}
```

This check is missing in the buy function.

**Recommendation**      Consider applying this check to the buy function as well.

**Resolution**      RESOLVED

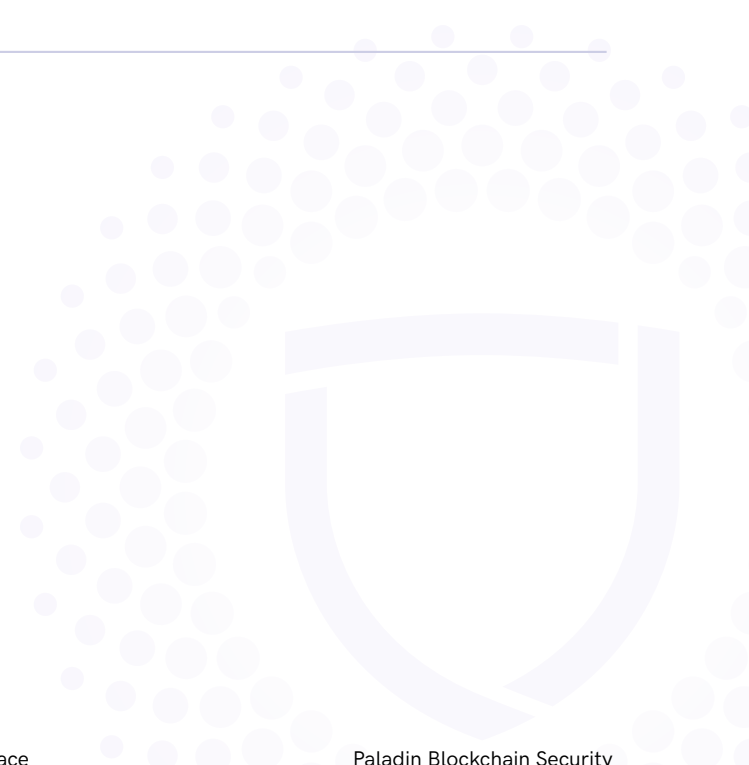
**Issue #72**      **Gas optimizations**

**Severity**      INFORMATIONAL

**Description**      \_registry can be marked as immutable.

**Recommendation**      Consider implementing the gas optimizations mentioned above.

**Resolution**      RESOLVED



**Issue #73****Typographical issues****Severity** INFORMATIONAL**Description**

Address, AggregatorV3Interface, IWETH9, IMarketBroker, IPortfolio, IPortfolioStorage, IPriceOracle, PortfolioAssetExtensions, Pausable, Sweepable and AuthorizedAccessControl are unused imports. Consider removing them.

---

\_registry should be marked as public to allow browser inspection by users.

---

It is unclear why the contract contains a receive function.

---

The contract should opt for forceApprove over safeIncreaseAllowance.

**Recommendation**

Consider fixing these issues.

**Resolution** RESOLVED

---

## 2.23 Marketplace/ SpotMarketMarketplaceAdapter

SpotMarketMarketplaceAdapter is a simple UniswapV2 aggregator where users can swap tokens using a variety of different Uniswap routers. Whenever a user executes a swap, the most optimal route from all routers is being fetched and the swap execution is being done on this router.

It should be noted that any permitted slippage, for example automatically via the liquidation mechanism, might get extracted through front-running the transaction. This is a design-choice made by the client and considered a trade-off.

It should finally be noted that we did not audit the underlying swap architecture and assume that the routing logic returns genuine values. Additionally, we assume that whichever DEX is used, swapExactTokensForTokens indeed provides the correct amount of tokens without doing malicious reentrancy logic.



## 2.23.1 Issues & Recommendations

**Issue #74**      **minAmountOut calculation for buy function normalizes incorrect decimals**

**Severity**

 HIGH SEVERITY

**Description**

Within the buy function, minAmountOut is calculated as follows:

```
uint minAmountOut = amount.mulDiv(10 ** decimals,  
price.normalize(decimals));
```

While this works if the highlighted decimals are in the same decimal denomination used for amount, it will not work if it is different.

Consider the following scenario:

- a) 100 USDT is swapped for a Portfolio token
- b) The portfolio token is denominated with 6 decimals
- c) The price is 1, respectively 100000000 as a Chainlink return value

This will result in the following calculation:

$$100e18 * 1e6 / 1e6$$

The output amount will therefore be 100e18 when it should be 100e6, effectively DoS'ing the whole functionality.

\*The same issue applies to the sell functiona:

```
uint minAmountOut = amount.mulDiv(price.normalize(decimals),  
10 ** decimals);
```

Let's say 1 ETH is swapped to USDT (6 decimals on mainnet), the calculation would be as follows:

$$1e18 * 2000e18 / 1e18 = 2000e18$$

Therefore, it would be expected that 2000e18 USDT will be received for 1 ETH, however, since USDT is in 6 decimals, this will revert always.

---

**Recommendation** Consider normalizing the denominator to the output token decimals:

```
uint minAmountOut = amount.mulDiv(10 ** decimals,
price.normalize(baseTokenDecimals));

uint minAmountOut =
amount.mulDiv(price.normalize(baseTokenDecimals), 10 **
decimals);
```

Note that this might cause truncation of the decimal values due to potential div-before-mul execution.

---

**Resolution**



---

**Issue #75**

**Inefficient architecture**

**Severity**



**Description**

The whole architecture relies on swaps using different UniswapV2 pairs. This might be inefficient in some cases and can easily result in high severity issues where swaps can be DoS'ed via an external user or simply will not work due to maximum hardcoded slippage on previous contracts within the transaction flow.

---

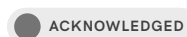
**Recommendation**

Consider simply abandoning the current architecture and switching to a DEX aggregator directly, 1Inch as example.

It should be noted that this comes with the downside of relying more on off-chain components and that slippage extraction for liquidations becomes guaranteed at that point, compared to a current possibility.

---

**Resolution**





**Issue #76****Truncation of decimals will result in price decrease****Severity** MEDIUM SEVERITY**Description**

Similar to the issue within DepositorVaultMarketPlaceAdapter, the truncation of 8 decimals to potentially 6 decimals can result in an undesired manipulation of minAmountOut:

L73

```
uint minAmountOut = amount.mulDiv(price.normalize(decimals),  
10 ** decimals);
```

L84

```
uint minAmountOut = amount.mulDiv(10 ** decimals,  
price.normalize(decimals));
```

**Recommendation**

A fix for this issue is non-trivial, as it would need a full refactoring. Do note that for complex fixes, there may be an extra fee levied to validate the code.

**Resolution** RESOLVED

The client has taken steps to avoid this premature truncation.



**Issue #77****Slippage application will result in higher slippage than expected****Severity** MEDIUM SEVERITY**Description**

Within `applyFeeAdjustment`, the slippage of the swap fee (0.3%) is first applied and then afterwards the provided slippage by the user is applied.

This will result in a larger slippage than was provided by the user, which can result in a relative loss depending on the swap amount.


Especially given the scenario that this estimation is used for the leverage and debt repayment scenario, a 0.3% slippage can already result in a large value.

**Recommendation**

Consider simply applying the provided slippage without the swap fee.

**Resolution** RESOLVED

The 0.3% swap fee has been used and only the user-provided input parameter is used as a slippage indicator. As discussed with the developer, the frontend should set a default minimum slippage of 0.5% as it is done with all DEXes.

**Issue #78****Hardcoded swapFee of 0.3%****Severity** LOW SEVERITY**Description**

Whenever a swap is executed, the 0.3% swap fee is applied to decrease the minimum output amount. However, not every DEX has a 0.3% swap fee, which can effectively mean:

- a) Higher unnecessary slippage
- b) DoS of swap due to a slippage which is too low

**Recommendation**

A fix is non-trivial since custom pairs do not have a standardized swap fee parameter. Consider simply switching to a completely different architecture if this is seen as an issue in practice.

Do note that for complex fixes, there may be an extra fee levied to validate the code.

**Resolution** ACKNOWLEDGED

## Severity

 INFORMATIONAL

## Description

The `IAddressRegistry`, `IDepositorVault`, `IMarket`, `IMarketStorage`, `IMarketLiquidation`, `IPortfolioStorage` and `Fees` imports appear to be fully unused. The `UnsafeSwapConditions` error appears to be unused as well.

---

`_routers`, `_baseToken` and `_token` should be marked as `public` to allow them to be inspected from within the browser. The latter two variables can also be marked as `immutable`.

---

Various address types can be cast to their explicit types such as `IUniswapV2Router01` and `IERC20Metadata`, straight into the storage section of the contract. This would avoid having to cast them later on and would be considered more explicit.

---

`forceApprove` is more appropriate than `safeIncreaseAllowance` within this contract.

---

`minAmountOut` is considered a misnomer within the swap arguments as this amount is reduced within the function body.

---

The `+ 2 minutes` addition within the swap is unnecessary and wastes gas. `block.timestamp` suffices.

---

The `findBestRoute` function should explicitly validate that a router was found, as it could occur that the catch clause is hit for every try attempt. A simple non-zero address requirement suffices at the end.

---

## Recommendation

Consider fixing the typographical issues.

---

## Resolution

 RESOLVED

---

## 2.24 Oracle/ChainlinkAggregatorPriceOracle

ChainlinkAggregatorPriceOracle is a simple helper contract that fetches the price from a Chainlink oracle. It is used within the following contracts:

- Liquidator
- Market
- Portfolio
- MarketLiquidation

It is assumed that this oracle is always behind a fallback price oracle to address staleness concerns. However, we have still communicated various potential extra safeguards with the client to improve their oracle redundancy as Paladin believes in defense-in-depth.

### 2.24.1 Issues & Recommendations

<b>Issue #80</b>	<b>Typographical issues</b>
<b>Severity</b>	<span>INFORMATIONAL</span>
<b>Description</b>	<p><u>Line 10</u> AggregatorV3Interface immutable private _aggregator;</p> <p>This variable can be marked as public to allow for users to inspect it within the explorer.</p>
<b>Recommendation</b>	Consider fixing the typographical issues.
<b>Resolution</b>	<span>RESOLVED</span>

---

## 2.25 Oracle/DepositorVaultTokenPriceOracle

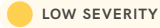

`DepositorVaultTokenPriceOracle` is a price oracle for the `VaultToken` which is received as receipt for any deposit within the `DepositorVault` contract. It represents the exact value of the underlying `baseAsset` using the `exchangeRate` in the `DepositorVault` contract. The contract also uses an arbitrary `_denominatorPriceOracle` to denominate the price.


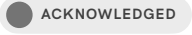
For example, `USDT` as `_denominatorPriceOracle` will denominate the oracle price as 1, while `AVAX` will denominate the price as 10 (assuming 1 `AVAX` = 10 `USD`).

The `_denominatorPriceOracle` should therefore always represent the same token as the underlying token of the `DepositorVault` contract, respectively `USDT` in `Ambit Finance's` architecture.



## 2.25.1 Issues & Recommendations

<b>Issue #81</b>	<b>Lack of staleness check for denominator oracle</b>
<b>Severity</b>	 LOW SEVERITY
<b>Description</b>	There is no staleness check when fetching the price from the denominator oracle which can lead to an outdated price.
<b>Recommendation</b>	Consider implementing such a check.
<b>Resolution</b>	 RESOLVED A boolean <code>isStale</code> is received when interacting with the denominator oracle — this boolean is then forwarded whenever a contract calls <code>getLatestPrice</code> .

<b>Issue #82</b>	<b>DepositorVaultTokenPriceOracle reports an incorrect price if the underlying token has a different number of decimals compared to the vault token</b>
<b>Severity</b>	 LOW SEVERITY
<b>Description</b>	<p>It appears that this contract assumes that the underlying token of the depositor vault must have an identical amount of decimals compared to the share token of that vault. This is however not enforced as of now.</p> <p>It should be noted that it makes sense for these decimals to be equal, since the depositor vault's shares are minted proportionally to the underlying token at first. This is the reason why this issue has only been raised as low, since it appears like these decimals will be equal in practice within the initial deployment.</p>
<b>Recommendation</b>	Consider enforcing this equality.
<b>Resolution</b>	 ACKNOWLEDGED

<b>Issue #83</b>	<b>Price is truncated</b>
<b>Severity</b>	<span style="color: yellow;">●</span> LOW SEVERITY
<b>Description</b>	<p>The price is calculated as follows:</p> <pre>uint price2 = exchangeRate.mulDiv(denominatorPrice.normalize(decimals), scalar).normalize(decimals, USDMath.DECIMALS);</pre> <p>However, since decimals are usually denominated in 18 for USDT on BSC, the result will be truncated to 8 decimals potentially decreasing the price. This can result in a lower price than desired.</p>
<b>Recommendation</b>	Consider if that is an issue, if yes, the logic needs to be adjusted.
<b>Resolution</b>	<span style="color: gray;">●</span> ACKNOWLEDGED

<b>Issue #84</b>	<b>Gas optimizations</b>
<b>Severity</b>	<span style="color: purple;">●</span> INFORMATIONAL
<b>Description</b>	_registry and _denominatorPriceOracle can be marked as immutable to save significant gas.
<b>Recommendation</b>	Consider implementing the gas optimizations mentioned above.
<b>Resolution</b>	<span style="color: green;">✓</span> RESOLVED





<b>Issue #85</b>	<b>Typographical issues</b>
<b>Severity</b>	<span>INFORMATIONAL</span>
<b>Description</b>	_registry and _denominatorPriceOracle can be marked as public to allow users to inspect their values from within the explorer.
<b>Recommendation</b>	Consider fixing the typographical issues.
<b>Resolution</b>	<span>RESOLVED</span>



---

## 2.26 Oracle/FallbackPriceOracle

FallbackPriceOracle is a custom implementation which fetches the price from two oracles:

- `_primaryOracle`
- `_fallbackOracle`

While the price is always first fetched from the primary oracle, if the staleness check does not succeed, it will switch to fetching the price from the fallback oracle. It can be used within the same contracts as the standard `ChainlinkAggregatorPriceOracle`.

It may make more sense to return the minimum between the two oracle prices, as it adds another layer of defense, though the client's advisor indicated that returning the least stale value is a design choice which was made.

Another recommendation which we have made to the client is to TWAP the price up if the oracle returns an abnormal price.

## 2.26.1 Issues & Recommendations

Issue #86

Oracle can be made magnitudes more secure within the current design by allowing the price to only slowly go up but go down instantly

Severity

LOW SEVERITY

Description

Principle: Oracle prices within the system seem to be mainly used for liquidation and collateral value to open positions.

Underestimating asset prices is safe for the protocol (somewhat risky for users), overestimating asset prices can be tremendously risky for the protocol.

The core principle is to underestimate prices for a safer protocol. If the system enforces that prices cannot suddenly increase by a large amount, the system is not exploitable in a short timeframe, regardless of how bad the oracles used are. This is insanely valuable for the security of the system, as long as the oracles are only used for collateral valuation.

This security principle allows you to implement simple but extremely strong safeguards, most notably:

1. Price goes instantly down, but TWAPs up: If an asset starts increasing in price, the oracle should catch up slowly instead of instantly, eg. 10%/hour might be a good heuristic. If the oracle price goes down however, it can go instantly down.
2. Taking the minimum price between multiple oracles. This is however a design choice since if an oracle malfunctions and returns price \$0, a lot of users may get liquidated.

Recommendation

Consider whether either design choice makes sense to seriously improve the safety of the oracle to a point where exploiting them becomes prohibitively difficult.

Resolution

ACKNOWLEDGED

**Issue #87**      **Staleness protection not guaranteed**

**Severity**      ● LOW SEVERITY

**Description**      Whenever the first oracle call returns a stale price, the second call is executed. However, if the second call also returns a stale price, the incorrect first stale price is used.

**Recommendation**      Consider reverting in that case.

**Resolution**      ● ACKNOWLEDGED  
Although a boolean is now propagated which indicates this.

**Issue #88**      **isStale could theoretically underflow and revert**

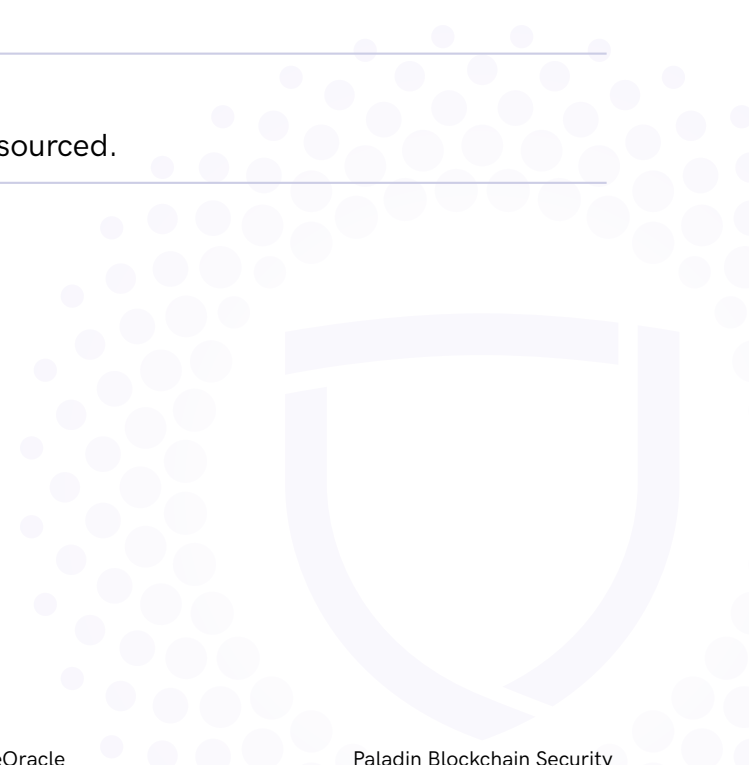
**Severity**      ● INFORMATIONAL

**Location**      Line 39  
`return block.timestamp - timestamp > _timeout;`

**Description**      If the original oracle malfunctions and returns a timestamp in the future, this underflows and reverts. Instead, it may make more sense to use the fallback oracle in this scenario, alongside the scenario where the primary Chainlink oracle reports a negative price.

**Recommendation**      Consider whether it makes sense to use a fallback oracle in such a scenario.

**Resolution**      ✓ RESOLVED  
The staleness check is outsourced.



<b>Issue #89</b>	<b>Gas optimizations</b>
<b>Severity</b>	<span>INFORMATIONAL</span>
<b>Description</b>	<code>_primaryOracle</code> , <code>_fallbackOracle</code> and <code>_timeout</code> can be marked as <code>immutable</code> to save significant gas.
<b>Recommendation</b>	Consider implementing the gas optimizations mentioned above.
<b>Resolution</b>	<span>RESOLVED</span>

<b>Issue #90</b>	<b>Typographical issues</b>
<b>Severity</b>	<span>INFORMATIONAL</span>
<b>Description</b>	<code>_primaryOracle</code> , <code>_fallbackOracle</code> and <code>_timeout</code> can be marked as <code>public</code> to make inspection by users through the explorer possible.
<b>Recommendation</b>	Consider fixing the typographical issues.
<b>Resolution</b>	<span>RESOLVED</span>



---

## 2.27 Portfolio/Custodian

Custodian is a simple vault contract which stores the asset from the Portfolio interactions. Each asset has its own vault and eventually can earn rewards based on the `LinearDistributionVault` logic.

At the current development phase, the Custodian will not receive any additional yield, however, the Ambit Finance team indicated that this will potentially change in the future.

The deposited collateral can potentially get liquidated.

### 2.27.1 Privileged Functions

- `pause`
- `resume`
- `sweep`
- `sweepETH`



## 2.27.2 Issues & Recommendations

<b>Issue #91</b>	<b>Checks-effects-interactions pattern is not adhered to, allows for the Portfolio <code>maxSupply</code> check to be bypassed through a reentrancy exploit</b>
------------------	---

**Severity** ● LOW SEVERITY

**Description** Within the `deposit` and `withdraw` function, the transfers are executed before the state variable changes happen.

This violates the CEI pattern and makes the contract vulnerable to reentrancy attacks.

It should be noted that the current design is consistent with the approach OpenZeppelin and other providers, where the interaction occurs at a state where neither the supply nor shares have increased yet.

The main reentrancy exploit we found which is however still possible is the ability to bypass the `asset.maxSupply` check within the Portfolio contract.

When checks-effects-interactions is not adhered to, finding all exploits is however a difficult task. We recommend adhering to the pattern instead to resolve this exploit and potentially future ones as well. A similar issue is present with the total asset check on withdrawals for now as well.

**Recommendation** Consider executing the effects before the interactions.

**Resolution** ✓ RESOLVED

Custodian has been refactored into a simple token storage contract, transfers are now done after state variable changes.

## Severity

INFORMATIONAL

## Description

Lines 8, 10, 21 and 23

```
import { IAddressRegistry } from "../../interfaces/core/
IAddressRegistry.sol";
import { AddressRegistryExtensions } from "../../libraries/
AddressRegistryExtensions.sol";
using AddressRegistryExtensions for IAddressRegistry;
IAddressRegistry private _registry;
```

These imports are unused and can be removed.

---

Being able to configure the initial total shares and assets at deployment is not only a governance risk, but also a configurational risk. Vaults are not supposed to be present with a state where the shares are non-zero but the underlying assets are zero. This state becomes possible due to this configurability. Consider either removing the constructor parameters to configure these values or explicitly requiring both values to be either zero or non-zero to avoid these non-permitted states.

## Recommendation

Consider fixing the typographical issues.

## Resolution

PARTIALLY RESOLVED

The initial total shares and assets are no longer directly configurable. The imports are left as-is.



**Issue #93****Gas optimizations****Severity**INFORMATIONAL**Description**Lines 58-60 and 77-79

```
if (amount == 0) {  
    return 0;  
}
```

These portions of code appear unused and unreachable due to the `requireNonZeroAmount` modifier and can therefore be removed.

Line 84

```
balance: getTotalAssets()
```

This was already fetched from storage and could be cached. Though arguably caching it might increase gas cost as this is an unlikely path so we are fine with this not being cached as well. This check in general however seems redundant with checks which could be (but are not right now, which we do not like) occurring within `decreaseTotalAssets`.

**Recommendation**

Consider implementing the gas optimizations mentioned above.

**Resolution**ACKNOWLEDGED

---

## 2.28 Portfolio/CustodianMigrator

CustodianMigrator facilitates a secure transfer of assets from an old custodian to a newly instantiated one. Initiated by an administrator, the contract pauses interactions with the old custodian, safely transfers assets to a temporary treasury, and then funnels them to the new custodian. Once transferred, system records are updated to reflect the new custodian, and permissions are strategically adjusted to maintain system security.

### 2.28.1 Issues & Recommendations

No issues found.

---

## 2.29 Portfolio/Portfolio

Portfolio is the entry point for users to supply and withdraw collateral. Users can supply any eligible asset to the Portfolio contract which then increases a user's share of this asset on a pro-rated basis depending on the total assets supplied, including the yield. The asset is transferred to the corresponding Custodian contract and serves as collateral for borrow activities within the Market contract.

This contract also includes functionality to determine the whole USD value of all positions from one specific address.

The authorized supply and withdrawal functionalities are used by the Marketplace contract during the buy and sell operations.



There is also a configuration risk where a bad custodian is configured for an asset, potentially pulling bad tokens from the user which they did not intend to. The team should be careful with configuring custodians and should keep their keys secure, as described further in the global governance issue.

*\*It should be noted that, if a user decides to supply assets via the Portfolio contract, the approval must be granted to the corresponding Custodian contract.*

### 2.29.1 Privileged Functions

- sweep
- sweepETH
- pause
- resume
- supply (Marketplace)
- withdraw (Marketplace)

## 2.29.2 Issues & Recommendations

<b>Issue #94</b>	<b>ETH liquidation is impossible due to special handling of ETH compared to WETH</b>
<b>Severity</b>	 HIGH SEVERITY
<b>Description</b>	<p>The ETH supply function uses exception logic within the Portfolio contract, thus it is managed differently to the WETH supplies. It uses a different configuration but also uses the exception hardcoded "0x0" asset to indicate that the supplied asset is the native gas token.</p> <p>By adding exception logic like this instead of simply relying on the existing ERC20 WETH logic, the complexity of the protocol is essentially doubled as throughout the protocol, every piece of code now needs to be aware that 0x0 is a special exception asset meaning "ETH tokens".</p> <p>This is done consistently throughout the protocol, except for one location we identified (there could be more): the liquidation procedures. These liquidation procedures rely on the asset address to represent the ERC20 token while for ETH it is 0x0. This completely prevents ETH liquidations from happening.</p>
<b>Recommendation</b>	<p>Consider not having exception logic like this for ETH; instead, rely on WETH for ETH deposits and withdrawals. The utility functions should simply wrap and unwrap into WETH and then call the ERC20 related code instead of using a magic 0x0 value.</p> <p>Do not fix the individual locations within liquidation, as this is only a symptom and not the cause of the disease.</p>
<b>Resolution</b>	 RESOLVED
	<p>The logic has been refactored to use WETH instead behind the scenes.</p>

**Description**

Within the `withdrawFrom` function, the call to the custodian is executed before the shares are decreased. As the custodian will transfer the ERC20 token to the receiver, this opens up a potential reentrancy state since at the point of the transfer, the user already has the funds but the portfolio storage still displays the old share amount of the user:

```
function withdrawFrom(address account, address receiver,
address asset, uint256 amount) private {
    IAssetStorage assetStorage =
_registry.getAssetStorage();

    IAssetStorage.Asset memory a =
assetStorage.getAsset(asset, ASSET_FIELD_CUSTODIAN);
    require(a.custodian != address(0));

    ICustodian custodian = ICustodian(a.custodian);

    IPortfolioStorage portfolioStorage =
_registry.getPortfolioStorage();

    IPortfolioStorage.PortfolioAsset memory portfolioAsset =
portfolioStorage.getPortfolioAsset(account, asset);

    uint shares = custodian.withdraw(receiver, amount);

    if (shares > portfolioAsset.shares) {
        revert Errors.Portfolio_WithdrawAmountTooLarge({
            amount: amount,
            shares: shares,
            totalShares: portfolioAsset.shares
        });
    }

    portfolioStorage.decreaseShares(account, asset, shares);
}
```

---

This will result in an un-updated state in favor of the user during the token transfer, where `getPortfolioValue` will return a larger value than the user actually has supplied.

---

**Recommendation** Consider strictly adhering to checks-effects-interactions throughout the codebase. Consider adding a global reentrancy guard if unsure.

*\*The same issue applies to `supplyTo`.*

---

**Resolution**

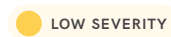


The share increase/decrease is now done before the token transfer.

---

## Issue #96 **Missing non-zero modifier**

**Severity**



**Description**

`withdrawETH` lacks a non-zero modifier for amount. This will potentially give users more flexibility to cause undesired function states.

---

**Recommendation** Consider adding such a modifier to this function.

---

**Resolution**



**Issue #97****Off-by-one error within supplyTo****Severity** LOW SEVERITY**Description**

Whenever a user supplies tokens, the following safety check is implemented:

```
if (portfolioStorage.getCount(account) >=
MAX_PORTFOLIO_ASSETS) {

    revert
    Errors.Portfolio_MaxmimumLimitReached(MAX_PORTFOLIO_ASSETS);
}
```

This basically ensures that a user cannot supply more than 20 assets. However, since the `increaseShares` call is made after the check, users can then reach the 20 threshold, which means that a withdrawal is needed to supply further.

**Recommendation**

Consider either checking the count post-increase, or keep the CEI structure and adjust the check itself.

**Resolution** RESOLVED

The check has been moved post-increase.

**Issue #98**      **Inconsistency between share increase/decrease**

**Severity**      ● LOW SEVERITY

**Description**      When supplying, the share increase is done as follows:  
`portfolioStorage.increaseShares(account, asset.token, shares);`

while during a withdrawal, the share decrease is done as follows:  
`portfolioStorage.decreaseShares(account, asset, shares);`

Within the decrease, the asset struct value is used while with the increase, the exact token for that asset struct is used.

This is not an issue since the invariant that asset value = asset.token is always given within the setAsset function in the AssetStorage contract. However, it should still be noted for consistency reasons.

**Recommendation**      Consider using asset.token for the decrease scenario as well.

**Resolution**      ✓ RESOLVED

**Issue #99**      **Additional layer of security: USD based caps**

**Severity**      ● INFORMATIONAL

**Description**      The Portfolio has absolute caps on the number of tokens that can be supplied in aggregate to a market. It may make sense to also cap the USD value of these assets.

**Recommendation**      Consider either USD based caps or implementing a recommendation as stated elsewhere where USD token value cannot increase with a high velocity.

**Resolution**      ● ACKNOWLEDGED



**Issue #100**      **Typographical issues**

**Severity**      

**Description**      L 6, 8 and 25  

```
import { IERC20Metadata } from "../../dependencies/  
@openzeppelin/contracts/token/ERC20/extensions/  
IERC20Metadata.sol";  
import { AggregatorV3Interface } from "../../dependencies/  
@chainlink/contracts/src/v0.8/interfaces/  
AggregatorV3Interface.sol";  
import { Normalizer } from "../../libraries/Normalizer.sol";
```

These imports are unused and can be removed.

---

\_registry should be made public here and within the other contracts.

**Recommendation**      Consider fixing the typographical issues.

**Resolution**      

**Issue #101**      **Gas optimizations**

**Severity**      

**Description**      \_registry can be marked as immutable here and within the other contracts.

---

weth can be cached.

**Recommendation**      Consider implementing the gas optimizations mentioned above.

**Resolution**      

---

## 2.30 Portfolio/PortfolioStorage

PortfolioStorage stores important state variables for the Portfolio contract such as which tokens the user has supplied and the corresponding shares for this token.

It should be noted that `getPortfolioAssets` may run out of gas and the client should be careful with not enabling too many assets for this reason.

### 2.30.1 Privileged Functions

- `increaseShares (Portfolio)`
- `decreaseShares (Portfolio)`

### 2.30.2 Issues & Recommendations

<b>Issue #102</b>	<b>Gas optimizations</b>
<b>Severity</b>	<span>INFORMATIONAL</span>
<b>Description</b>	<p><u>Line 63</u> <code>if (portfolioAsset.shares == 0) {</code></p> <p>This value has already been fetched at this point, it can be cached to save gas.</p>
<b>Recommendation</b>	Consider implementing the gas optimizations mentioned above.
<b>Resolution</b>	<span>RESOLVED</span>

---

## 2.31 Security/AdminAccessControl

`AdminAccessControl` is a simple access control dependency to manage an admin role for contracts using this dependency which can be granted to several accounts. It expands upon OpenZeppelin's `AccessControl` contract and introduces a single modifier which can be used to guard functions: `onlyAdmin`.

It is used within the following contracts:

- `AddressRegistry`
- `Overseer`
- `Pausable`



The privileged role is stored as `ADMIN_ROLE` with `0xdf8b4c520ffe197c5343c6f5aec59570151ef9a492f2c624fd45ddde6135ec42` as the role key.

### 2.31.1 Privileged Functions

- `renounceRole [ role bearer ]`
- `grantRole`
- `revokeRole`



## 2.31.2 Issues & Recommendations

<b>Issue #103</b>	<b>Unused import</b>
<b>Severity</b>	 INFORMATIONAL
<b>Location</b>	<u>L5</u> <code>import { AuthorizedAccessControl } from "../AuthorizedAccessControl.sol";</code>
<b>Description</b>	This import is unused.
<b>Recommendation</b>	Consider removing the unused import.
<b>Resolution</b>	 RESOLVED



---

## 2.32 Security/AuthorizedAccessControl

AuthorizedAccessControl is similar to the AdminAccessControl contract. It is used within the following contracts:

- AssetStorage
- Treasury
- MarketLiquidation
- MarketStorage
- Custodian
- PortfolioStorage

The privileged role is stored as AUTHORIZED\_ROLE with `0x50e75b23f8ec51bbcb044fe377457e835f3b5ccc7ebf69e80906452015dff607` as the role key.

### 2.32.1 Issues & Recommendations

No issues found.



---

## 2.33 Security/AccessControlList

`AccessControlList` provides a framework for role-based access control. Upon initialization, the deployer is assigned the `ADMIN_ROLE`. This role is pivotal as it can grant or revoke roles to other addresses through the `grantRole` and `revokeRole` functions respectively.

Any address can willingly renounce its role via the `renounceRole` function. A safety measure is in place to ensure there is always at least one admin — the last `ADMIN_ROLE` cannot be removed.

Events such as `RoleGranted` and `RoleRevoked` ensure transparency by signalling role changes.

A specialized `onlyAdmin` modifier restricts certain functions only to admin access such as `grantRole` and `revokeRole`. `hasRole` function is used as a pivotal guide throughout `Ambit`'s whole infrastructure, ensuring a smooth and secure access control mechanism.

### 2.33.1 Issues & Recommendations

No issues found.



---

## 2.34 Tokens/AmbitToken

AmbitToken is a simple token using LayerZero's Omichain Fungible Token implementation (<https://github.com/LayerZero-Labs/solidity-examples/tree/main>) with 18 decimals.

The token includes a burn functionality which allows the caller to burn their own token balance, however, it is restricted to be called by the Treasury contract only.

The contract mints an initial supply of 100\_000\_000e18 tokens to the Treasury address.

### 2.34.1 Privileged Functions

- burn (treasury)



## 2.34.2 Issues & Recommendations

<b>Issue #104</b>	<b>Typographical issues</b>
<b>Severity</b>	<span>INFORMATIONAL</span>
<b>Description</b>	<p><u>Line 14</u> error NotPermitted(address caller);</p> <p>The contract could re-use Authorization_NotAuthorized error from Errors.sol here.</p> <p><u>Line 16</u> IAddressRegistry immutable private _registry;</p> <p>_registry should be made public to allow for users and reviewers to easily inspect it in the explorer.</p>
<b>Recommendation</b>	Consider fixing the typographical issues.
<b>Resolution</b>	<span>ACKNOWLEDGED</span>





---

## 2.35 Utils/Pausable

Pausable implements pausing and unpausing functionality and is used throughout the ecosystem within the following contracts:

- Treasury
- Market
- MarketLiquidation
- Custodian
- Portfolio
- DepositorVault
- DepositorVaultToken

### 2.35.1 Privileged Functions

- pause
- resume

### 2.35.2 Issues & Recommendations

No issues found.

---

## 2.36 Utils/Sweepable

Sweepable implements sweeping functionality such as transferring ERC20 token and ETHER out of the contract. It is used throughout the ecosystem within the following contracts:

- Liquidator
- Market
- Portfolio

Note that the Sweepable dependency does not revert if the contract has insufficient tokens. In this case, it simply transfers whatever balance it has to the treasury.

### 2.36.1 Privileged Functions

- sweep
- sweepETH
- renounceRole [ role bearer ]
- grantRole [ DEFAULT\_ADMIN\_ROLE ]
- revokeRole [ DEFAULT\_ADMIN\_ROLE ]




## 2.36.2 Issues & Recommendations

Issue #105

**Sweeping Ether to treasury might not work for treasuries that execute logic on their fallback**

Severity

 LOW SEVERITY

Description

Since the treasury is a contract, transfer will revert due to only forwarding 2300 gas with the call if the recipient executes code in its fallback. This will result in stuck Ether in the contract if the treasury address is changed to one with fallback logic within the registry contract.

Additionally, there is no guarantee that the gas costs will not change over time with hard-forks, making this extra risky.

Recommendation

Consider using `.call` instead of `.transfer`.

Resolution

 RESOLVED

**Issue #106** **Typographical issue**

**Severity** INFORMATIONAL

**Description** L4 and 17  
`import { Address } from "../../dependencies/@openzeppelin/contracts/utils/Address.sol";  
using Address for address payable;`

These imports appear unused.

L19  
`IAddressRegistry private _registry;`

This should be marked as public.

L27  
`function sweep(address token, uint amount) external  
onlyAuthorized {`

token can be directly provided as type IERC20.

**Recommendation** Consider fixing the typographical issues.

**Resolution** RESOLVED

**Issue #107** **sweepETH is payable**

**Severity** INFORMATIONAL

**Description** This function is payable, however, there is no reason to send any ETH with this transaction.

**Recommendation** Consider removing the payable keyword.

**Resolution** RESOLVED

---

## 2.37 Vault/DepositorVault

DepositorVault is a simple vault contract where users can deposit an asset and receive the vaultToken as position receipt. The underlying value of this position will then increase due to various different mechanisms of which all are based on the donate functionality.

The admin can set a borrow limit for any arbitrary address which then allows this address to borrow tokens from the vault without a collateral position for an arbitrary fee. It is primarily intended that the Market contract is the borrowing party. The borrow limit can be either expressed as absolute value or as basis points. In the latter case, the basis point value will be applied on the total current available balance which is denominated as  $(totalAssets - totalLiabilities)$ , including distributed yield up to the current timestamp. The borrowable amount is then calculated as  $theoreticalLimit - liabilities$ .

The donation functionality allows yield to accumulate which is then linearly distributed over the course of the next 6 hours. Yield will be generated from the interest which users pay on their borrowing positions as well as from a part of the profit which is generated during liquidations.

It should be noted that the borrower's utilization rate can exceed 100% in cases like when the limit is reduced. This has an impact on the interest rate, which could become higher than expected for users. But this appears to be a design decision so it was not added as an issue.

The contract is not compliant with deposit tokens that have a fee on transfer — this is fine as the USDT implementation the client plans to use will realistically never have such functionality. Additionally, the client has indicated that they do not ever plan to support a reentrancy deposit token.

## 2.37.1 Privileged Functions

- `setReserveBPS` [ ADMIN\_ROLE ]
- `setFlashloanFee` [ ADMIN\_ROLE ]
- `setMaxSupply` [ ADMIN\_ROLE ]
- `setBorrowLimit` [ ADMIN\_ROLE ]
- `sweep` [ AUTHORIZED\_ROLE ]
- `sweepETH` [ AUTHORIZED\_ROLE ]
- `pause` [ ADMIN\_ROLE ]
- `resume` [ ADMIN\_ROLE ]
- `renounceRole` [ role bearer ]
- `grantRole` [ DEFAULT\_ADMIN\_ROLE ]
- `revokeRole` [ DEFAULT\_ADMIN\_ROLE ]



## 2.37.2 Issues & Recommendations

Issue #108

**DoS exploit: Several functions including `getAvailableBalance` can be exploited when utilization is 100% by withdrawing manually sent USDT, causing these functions to brick and revert due to an underflow vulnerability**

Severity



Location

Line 125

```
uint256 balance = getTotalAssets() - getTotalLiabilities();
```

Description

Several sections of code within the `DepositorVault` calculate the unallocated balance as the difference between the vault's assets and the vault's liabilities. Assets are increased whenever people call the `deposit` function to deposit USDT into the vault, and decreased whenever they withdraw USDT via the `withdraw` function. Liabilities are similarly increased and decreased through borrow calls and repay calls.

Once all USDT is borrowed out, given a reserve factor of 0, the balance of the vault becomes 0. This prevents any further withdrawals simply because the vault does not have any USDT to send to the withdrawing user.

This is good because if people could still make withdrawals, line 125 would underflow and various sections of code would suddenly start reverting, causing the contract to severely malfunction.

However, it turns out that there is still a clever way for an exploiter to instantiate a withdrawal. The exploiter could in fact send some USDT manually to the contract, through a regular transfer, and then withdraw it again.

Though this has no direct financial benefit, it would cause `getTotalAssets()` to decrease on the withdraw, without these assets increasing when the USDT was sent. This causes line 125 to underflow and multiple sections of the contract to become bricked. The exploiter has therefore succeeded at denying service to multiple portions of the contract by abusing the lack of accounting checks within the `withdraw` function, which instead uses the USDT balance as the check to ensure that withdrawals are permitted.

---

**Recommendation** Consider not relying on the USDT balance to validate whether withdrawals and redemptions should be allowed. Instead, validate the accounting parameters (`getTotalAssets()` - `getTotalLiabilities()`) explicitly.

---

**Resolution**



A check has been introduced within the `withdraw` function, ensuring that the withdrawal amount cannot be larger than the difference of deposited assets and liabilities.

---





**Description**

The utilization ratio is computed as follows:

```
Math.mulDiv(liabilities.normalize(decimals, 27),  
RayMath.RAY, effectiveLimit.normalize(decimals, 27));
```

However, if the `effectiveLimit`, which is either the `theoreticalLimit` or the amount of assets within the contract, is zero, the whole utilization value becomes zero to prevent a division by zero:

```
uint256 effectiveLimit = Math.min(theoreticalLimit,  
getTotalAssets());
```

```
uint utilization = effectiveLimit == 0  
? 0
```

Due to the nature of lending pools, the balance will never cover the whole supplied amount of depositors, hence one single depositor may force the vault into a state, where no assets are left in the vault with a simple withdrawal:

```
function decreaseTotalAssets(uint amount) internal {  
    (uint totalAssets, uint totalYield) = distributeYield();  
  
    _totalAssets = amount > totalAssets ? 0 : totalAssets -  
amount;  
    _totalYield = totalYield;  
    _distributionTimestamp = block.timestamp;  
}
```

The depositor has now artificially caused a utilization rate of zero, resulting in a manipulation of the `borrowIndex` within the Market contract.

Another way to increase/decrease the current utilization ratio is to simply execute a large deposit/withdrawal before the market state update, as this will impact the utilization ratio, since `borrowLimit` is based on the total amount of assets deposited:

---

```
BPS.wrap(uint16(borrowLimit.amount)).percentOf(getTotalAssets());
```

A very large deposit can therefore artificially increase the `borrowLimit` to a minimum if the relative approach is activated. This can be constantly used to front run every market state update, since the deposited amount can be withdrawn immediately again.


The `_reserveBPS` checks within `getAvailableBalance` can also be bypassed through a similar flash exploit.

---

**Recommendation** Consider updating the market state before each deposit/ withdrawal. Consider not overly relying on instantaneous relative balance checks on the depositor vault. Checks like the `reserveBPS` and relative borrow limits can presently be trivially bypassed after all.

---

**Resolution**

 ACKNOWLEDGED

The client has indicated they are solving this in further iterations using a weighted average of balance snapshots.

---



## Severity

 HIGH SEVERITY

## Description

At the current development state, it is only intended to allow borrowing/repaying for the Market contract. However, in the future this might change, and any of the authorized addresses can call the repay function to repay borrowed funds.

However, this function is flawed in the scenario where the caller submits a larger amount than the actual debt — in this case, the amount is being transferred from the caller while the maximum liabilities are decreased:

```
function repay(uint256 amount) external onlyAuthorized
whenNotPaused requireNonZeroAmount(amount) {
    IERC20Metadata underlyingAsset = getUnderlyingAsset();

    uint256 balance = underlyingAsset.balanceOf(msg.sender);

    if (amount > balance) {
        revert Errors.DepositorVault_AmountExceedsBalance({
            amount: amount,
            balance: balance
        });
    }

    IDepositorVaultStorage vaultStorage = getVaultStorage();

    // the borrower can repay more than their liabilities
    // if they are paying back yield to the pool
    uint256 liabilities = Math.min(amount,
    vaultStorage.getLiabilities(msg.sender));

    vaultStorage.decreaseLiabilities(msg.sender,
    liabilities);
    underlyingAsset.safeTransferFrom(msg.sender,
    address(this), amount);
    emit Repay(msg.sender, amount);
}
```

This will result in a loss of tokens for the caller.

## Recommendation

Consider requiring the amount parameter to be smaller or equal to liabilities instead.

---

## Resolution



The amount is now silently reduced to the liabilities of the sender. It should be noted that this is done silently which might confuse callers. Integrations should be very careful with this behavior.

We remind the client to carefully validate this for all current borrowing contracts and for all future ones as well, as this could be a mistake waiting to happen if not carefully checked.

---



## Severity

 MEDIUM SEVERITY

## Description

Whenever a flashloan is carried out, a fee is applied based on the `_flashloanFee` and the size of the flashloan:

```
uint fee = flashFee(token, amount);
```

```
function flashFee(address token, uint256 amount) public view
returns (uint256) {
    if (token != address(getUnderlyingAsset())) {
        revert
        Errors.DepositorVault_FlashLoanTokenNotSupported(token);
    }
    return _flashLoanFee.calculate(amount);
}
```

This fee is then included during the repay of the flashloan:

```
underlyingAsset.safeTransferFrom(address(receiver),
address(this), amount + fee);
```

However, unlike traditional vaults, this vault does not rely on the contract balance for the `totalAssets` variable, hence the fee will not increase the vault value, essentially being stuck forever in the contract.

***\*This issue was downgraded from high to medium severity because the vault inherits the Sweepable contract in the newest commit.***

## Recommendation

Consider calling a new `donateInternal` with the fee amount which does not transfer in the amount as it is already within the contract. Alternatively as discussed with the client, it may make more sense to remove the flashloan logic from the core altogether and modularize it into an authorized borrower.

## Resolution

 RESOLVED

The flashloan logic has been outsourced to the `FlashLender` contract and the fee is donated to the vault.

## Severity

 MEDIUM SEVERITY

## Description

The contract implements a `_reserveBPs` variable which reflects how much of the balance (`getTotalAssets() - getTotalLiabilities();`) should be kept in the contract as reserve.

Unfortunately, this security mechanism can be circumvented by borrowing assets in steps to reduce the balance (and adding them as assets temporarily for example).

Since the total balance can be decreased to a total minimum (depending on the `borrowLimit`), the reserve value can artificially be decreased to almost zero.

## Recommendation


Consider refactoring the reserve logic completely. Please note that if the change is significant, we may need to charge a revalidation fee for it.

## Resolution

 PARTIALLY RESOLVED

The client has increased the strictness of this check by requiring a portion of the total supplied assets to be reserves instead. However, it can still be bypassed using a large deposit.

## Severity

 MEDIUM SEVERITY

## Description

Whenever the `flashLoan` function within the `DepositorVault` is called, the desired amount is transferred out and the callback function of the recipient contract is called.

However, at this point, the `DepositorVault` is in an un-updated state since the liabilities are not increased nor the total assets are decreased (the latter is perfectly fine and mandatory to not decrease).

Due to this fact, several functions including `getBorrowableAmount` and `getAvailableBalance` return incorrect values which can potentially lead to undesired side-effects.

## Recommendation

Consider implementing a mechanism that prevents this un-updated state, i.e., increasing the liabilities before the transfer out and decreasing them again after the callback was successful.

However, it should be noted that updating state during a flashloan like this may do more harm than good. It is a design decision whether these loans should be considered proper borrows or not, and we leave that decision to the client.

During discussions with the client, it appeared to make sense to modularize out the flashloan logic into a separate borrower for this contract. This would make this logic a proper borrower and accounting variables would get updated. However, it would also reduce the attack surface of this contract as no internal flashloan possibilities exist within it. We agree with such an externalization but have recommended the client to do so with extreme care, as new interactive exploits might become possible if not done carefully.

## Resolution

 RESOLVED

The flashloan logic has been removed from the contract.

**Issue #114**      **Change of borrowLimit will change market state retroactively**

**Severity**      MEDIUM SEVERITY

**Description**      Since borrowLimit impacts the getBorrowerUtilization function which is used to determine the current rate, a change of the this variable will change the market state retroactively.

**Recommendation**      Consider updating the market state before the borrowLimit change.

**Resolution**      ACKNOWLEDGED

**Issue #115**      **Architectural flaw: Initial positions will not receive any yield**

**Severity**      MEDIUM SEVERITY

**Description**      The main yield generation process happens during the repayment of an outstanding borrow position. The donate functionality will distribute the paid interest over time.

However, initial depositors will not receive any yield on their positions until the creditors repay their open borrow position which essentially allows other users to frontrun the repay call to immediately gain interest on their newly staked position.

This will more and more become an issue if the platform scales and events such as the potential large curve liquidation could happen. Users could abuse these scenarios to only deposit shortly before the liquidation and receive yield for the next six hours.

**Recommendation**      Consider creating additional incentives to distribute yield for the initial depositors.


**Resolution**      ACKNOWLEDGED

The client has indicated they understand this as a design limitation and plan to keep a careful eye through for example interest rate management.



<b>Issue #116</b>	<b>Uncapped flashloan fee</b>
<b>Severity</b>	<span style="color: yellow;">●</span> LOW SEVERITY
<b>Description</b>	While we acknowledge that this architecture relies on full governance privileges, it still might be desired to cap the upper limit of sensitive variables like the <code>_flashLoanFee</code> .
<b>Recommendation</b>	Consider setting a reasonable upper limit for the variable if desired. Since flashloaning is not a core functionality, leaving the fee uncapped is also fine with us.
<b>Resolution</b>	<span style="color: green;">✓</span> RESOLVED This logic has been removed from this contract altogether. Moreover, the maximum flashloan fee was set to 10% within the FlashLender.

<b>Issue #117</b>	<b>borrowLimit can be set to be less than liabilities</b>
<b>Severity</b>	<span style="color: yellow;">●</span> LOW SEVERITY
<b>Description</b>	<p><code>setBorrowLimit</code> allows an absolute rate to be set, however, this rate can even be below the current liabilities, eventually manipulating the utilization ratio to a large value.</p> <p><u>Line 176</u>  <code>BPS.wrap(uint16(borrowLimit.amount)).percentOf(getTotalAssets());</code></p> <p>This portion of code would overflow in case the borrow limit is set too high. It should be noted that the relative borrow limit can be bypassed through flash supplies at this moment.</p>
<b>Recommendation</b>	Consider if this is desired, if not consider only allowing an absolute limit up to the current liabilities. Consider capping the relative <code>borrowLimit</code> up to 100% BPS or at least use <code>SafeCast</code> .
<b>Resolution</b>	<span style="color: grey;">●</span> ACKNOWLEDGED No changes to the borrow settings limit have been made.

**Issue #118****getBorrowerUtilization will return 1e27 for limit of zero****Severity** LOW SEVERITY**Description**

Whenever an address has a borrowLimit of zero, it cannot borrow any funds (ideally), however, this function still returns a utilization ratio of 1e27 (100%) which could be considered inconsistent in that scenario (since at a later stage 0 is returned in case the effective limit is zero).


**Recommendation**

Consider whether this makes sense and consider what variable would be more appropriate to return in such cases. Consider being consistent with this. Perhaps it makes sense to only return 1 in case the account has in fact borrowed anything.

**Resolution** RESOLVED

The client has indicated they carefully validated this and indeed prefer returning 0 in this case.



**Issue #119****Lack of checks-effects-interactions allows the maxSupply check to be bypassed if the token is vulnerable to re-entrancy****Severity** LOW SEVERITY**Location**

```
getUnderlyingAsset().safeTransferFrom(msg.sender,
address(this), amount);

increaseTotalAssets(amount);

// we need to do the transfer before we mint so that any
// reentrancy would happen before the
// assets are transferred and before the shares are minted,
// which is a valid state
IDepositorVaultToken vaultToken = getToken();
vaultToken.mint(receiver, shares);

emit Deposit(msg.sender, receiver, amount, shares);

return shares;
```

**Description**

Within the deposit function, transfer is executed before the increase of the total assets. This violates the CEI pattern and makes the function vulnerable to reentrancy attacks.

\* The same issue applies to `withdrawInternal`.

An example exploit that could occur is that the `maxSupply` check is bypassed during a reentrancy call as `totalAssets` would have yet to be updated at the point of reentrancy, allowing for this check to pass even if the outer deposit would cause it to fill up fully.

This should however only be considered as an example exploit. When code is not strictly written within checks-effects-interactions, the potential for exploits can be limitless and it is hard to enumerate all of them. We therefore strongly urge the client to adhere strictly to checks-effects-interactions.

**Recommendation**

Consider executing the transfer after the state variable change.

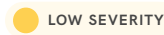
Consider refactoring the flashloan logic to be more careful as it does not presently adhere to checks-effects-interactions. No exploits were found within this section as of present however. Do note that for significant code changes in the resolution round, a revalidation fee may apply.

---

**Resolution**

The transfer now happens at the very end of this function.

---

**Issue #120****The reserve assets cannot be flashloaned****Severity****Description**

Within the flashloan function, up to the maxFlashloan can be borrowed. This function simply returns the `getAvailableBalance` which is the difference between assets and liabilities, with the reserve subtracted from it.

We do not see a reason why the reserve should not be borrowable here, since these flashloans will be paid back instantly anyway.

**Recommendation**



Consider whether it makes sense to increase the flashloan portion to `assets - liabilities` instead.



**Resolution**

The flashloan function has been fully removed from this contract. Note that this will still not be possible in the new flashloan design, but given that it will not be easily solvable in that design, we have marked this issue as resolved now.

---



<b>Issue #121</b>	<b>Gas optimizations</b>
<b>Severity</b>	
<b>Description</b>	<p><u>Line 93</u></p> <pre>function setFlashLoanFee(Fees.Parameters memory fee) external onlyAdmin {</pre> <p>The fee can be provided as calldata instead.</p>
<b>Recommendation</b>	Consider implementing the gas optimizations mentioned above.
<b>Resolution</b>	 This logic has been removed.

<b>Issue #122</b>	<b>Typographical issues</b>
<b>Severity</b>	
<b>Description</b>	<p><u>Line 4</u></p> <pre>import { IERC3156FlashLender, IERC3156FlashBorrower } from "../../dependencies/@openzeppelin/contracts/interfaces/ IERC3156.sol";</pre> <p>This import is unused and can be removed.</p> <p><u>Line 39</u></p> <pre>IAddressRegistry private immutable _registry;</pre> <p>This variable should be marked as public to allow for easier inspection within the browser.</p>
<b>Recommendation</b>	Consider fixing the typographical issues.
<b>Resolution</b>	

---

## 2.38 Vault/DepositorVaultMigrator

DepositorVaultMigrator is a simple helper contract that migrates funds from an old DepositorVault to a new DepositorVault contract. It must be ensured that the same storage contract is being used in order to keep all necessary deposits and borrows valid.

It is necessary that the migrator upfront has the mandatory privileges to execute all desired functionalities.



We recommend that the client carefully simulate all migrations and ideally migrate to dummy contracts and redeploy with a fresh state, since migrating using this mechanism is overall risky due to states being reset.

### 2.38.1 Privileged Functions

- migrate



## 2.38.2 Issues & Recommendations

<b>Issue #123</b>	<b>Yield will be lost with new vault</b>
<b>Severity</b>	 HIGH SEVERITY
<b>Description</b>	<p>The whole balance is transferred to the new vault, including the outstanding yield. However, the new vault's <code>_totalYield</code> state variable is set to zero, which effectively results in a loss of the pending yield for all users and results in unused funds within the new <code>DepositorVault</code>.</p> <p>This issue is rated as high severity since we assume that these funds will be left in the contract without anyone noticing. We acknowledge that it is possible via admin interactions to withdraw and donate them manually.</p>
<b>Recommendation</b>	Consider explicitly donating the current <code>_totalYield</code> .
<b>Resolution</b>	 RESOLVED
	<p>A <code>distribute</code> call is now made which incorporates the pending yield into the relevant accounting variables. These accounting variables can then be forwarded to the new contract using the new migration context.</p> <p>We recommend the client however to still be extremely careful with any such migrations and carefully simulate them in a test environment first. Especially since Paladin would not have audited any of the contracts the client would migrate to.</p>

## Severity

 MEDIUM SEVERITY

## Description

The following state variables are zero within the new DepositorVault contract:

- a) `_reserveBPS`: A malicious user can backrun the migration call and circumvent the previous set reserve, effectively emptying the whole contract
- b) `_maxSupply`: Users can backrun the migration call and circumventing the said limit
- c) `_flashloanFee`: Users can execute a leverage transaction without paying a flashloan fee.

## Recommendation

Consider setting these variables during the migration call.

## Resolution

 RESOLVED

The client has introduced a configurable migration context that can incorporate these variables alongside future variables. As always, we recommend the client to exercise utmost care with migrations and simulate them carefully.

It is important to note that the newly introduced `whenNotInSameBlockAsDeposit` modifier can be circumvented in the same block a migration happens.



**Issue #125****Swept funds are sent to the wrong vault****Severity** MEDIUM SEVERITY**Description**

After sweeping the funds to the treasury while migrating the vault, those funds should be sent to the new deployed vault as can be seen in the following snippet:

```
ITreasury treasury = _registry.getTreasury();  
treasury.transfer(address(underlyingAsset),  
address(depositorVault), balance);
```

However, the funds are actually sent back to the old vault instead of the new one.

**Recommendation**

Consider sending the swept funds to the new vault instead of the old one:

```
ITreasury treasury = _registry.getTreasury();  
treasury.transfer(address(underlyingAsset),  
address(depositorVaultUpgrade), balance);
```

**Resolution** RESOLVED

**Issue #126**      **Typographical issues**

**Severity**      INFORMATIONAL

**Description**      L11  
`import { AddressRegistryExtensions } from "../../libraries/AddressRegistryExtensions.sol";`

This import is a duplicate and can be removed.

L85  
`function revokeDefaultAdmin(address source, address dest) private`

L100  
`function revokeAdmin(address source, address dest) private`

These functions are unused and can be removed.

**Recommendation**      Consider fixing the typographical issues.

**Resolution**      RESOLVED

**Issue #127**      **Vendor and Purchaser are still granted rights**

**Severity**      INFORMATIONAL

**Description**      During the migration, both contracts are granted the auth status. However, since the FlashLender logic is not outsourced, there is no necessity to grant this status to both contracts.

**Recommendation**      Consider removing that logic.

**Resolution**      RESOLVED

---

## 2.39 Vault/DepositorVaultStorage

DepositorVaultStorage keeps track of the borrow limits and liabilities for the DepositorVault contract.

Slight gas optimizations are possible by marking certain math operations as unchecked but Paladin did not explicitly create issues for these due to the non gas-critical nature of the clients deployment chain.

### 2.39.1 Privileged Functions

- `setBorrowLimit`
- `increaseLiabilities`
- `decreaseLiabilities`

### 2.39.2 Issues & Recommendations

No issues found.



---

## 2.40 Vault/DepositorVaultToken

DepositorVaultToken is the ERC20 token receipt which will be minted on each deposit in the DepositorVault contract. It represents the amount a user has staked into the DepositorVault, including the accumulated yield.

It can be also used within the Portfolio as collateral, representing the said share value.

### 2.40.1 Privileged Functions

- pause
- resume
- mint (only vault)
- burn (only vault)

### 2.40.2 Issues & Recommendations

No issues found.



---

## 2.41 Vault/LinearDistributedYieldVault

LinearDistributedYieldVault is a yield-bearing implementation which is used for the Custodian and DepositorVault contract.

It allows for token donation to the vault via the donate function and the donated amount is then distributed over the course of 6 hours to all stakers by increasing the `_totalAssets` variable.




## 2.41.1 Issues & Recommendations

Issue #128

Non-configurable `_distributionWindow` may backfire in the future

Severity

 LOW SEVERITY

Description

`_distributionWindow` is initialized in the constructor and cannot be updated after.

This value should have a setter function to be updated only by `Ambit`, if necessary. Having a fixed `_distributionWindow` might be not the optimal in the future and in the current state, there should be a re-deployment to update it

Recommendation


Consider adding a function which also distributes the Yield before updating the window, so the new window does not affect the unclaimed yield:

```
function setDistributionWindow(uint256 distributionWindow)
external onlyOwner {
    (uint totalAssets, uint totalYield) = distributeYield();
    _totalAssets = totalAssets;
    _totalYield = totalYield + amount;
    _distributionTimestamp = block.timestamp;

    _distributionWindow = distributionWindow;
}
```

Resolution

 ACKNOWLEDGED

**Issue #129****The distribution rate is counter-intuitively non-linear depending on when `distributeYield` gets called****Severity** LOW SEVERITY**Description**

Due to the design pattern of the `LinearDistributedYieldVault`, the actual distribution rate is in fact non-linear. This is because whenever yield is distributed, the remaining yield is re-compounded over the distribution window. This means that even though this remaining yield was supposed to be fully distributed in a timespan quicker than the window at the time of an intermediary distribution, it gets reset to a duration equal to the window.

This has the effect that any yield distribution directly impacts the distribution rate for the remaining tokens yet to be distributed.

**Recommendation**

This is a fundamental design trade-off. We do not have a recommendation as solutions which adhere to a real linear distribution might require expensive iterative logic.

**Resolution** ACKNOWLEDGED

**Issue #130****Lack of checks-effects-interactions****Severity** INFORMATIONAL**Description**

Within the donate function, transfer is executed before the effects. This violates the CEI pattern and can result in reentrancy vulnerabilities:

```
function donate(uint256 amount) virtual public {
    if (amount == 0) {
        return;
    }

    getUnderlyingAsset().safeTransferFrom(msg.sender,
address(this), amount);

    (uint totalAssets, uint totalYield) = distributeYield();

    _totalAssets = totalAssets;
    _totalYield = totalYield + amount;
    _distributionTimestamp = block.timestamp;

    emit Donate(msg.sender, amount);
}
```

**Recommendation**

Consider executing transfer after the effects

**Resolution** RESOLVED



<b>Issue #131</b>	<b>Gas optimizations</b>
<b>Severity</b>	<span>INFORMATIONAL</span>
<b>Description</b>	<p><code>_distributionWindow</code> should be marked as <code>immutable</code>.</p> <p>---</p> <p>The <code>distributeYield</code> logic re-uses a lot of storage variables, reading from them multiple times and wasting some gas.</p>
<b>Recommendation</b>	Consider implementing the gas optimizations mentioned above.
<b>Resolution</b>	<span>RESOLVED</span>

<b>Issue #132</b>	<b>Typographical issues</b>
<b>Severity</b>	<span>INFORMATIONAL</span>
<b>Description</b>	<p>"elapsed" is mis-spelled as "ellapsed" throughout the contract.</p> <p>---</p> <p><code>_distributionWindow</code>, <code>_totalAssets</code> and <code>_distributionTimestamp</code> should be marked as <code>public</code>.</p> <p>---</p> <p>A <code>div-before-mul</code> occurs within <code>distributeYield</code> which slightly lowers precision, this is however fine in this instance as precision is not critical.</p>
<b>Recommendation</b>	Consider fixing the typographical issues.
<b>Resolution</b>	<span>ACKNOWLEDGED</span>

---

## 2.42 Vault/Vault

Vault handles the share calculation logic for DepositorVault and Custodian. It also returns the current exchange rate which displays the amount of assets per share, denominated in the decimals of the underlying asset.

For example, for USDT as an underlying asset, an exchange rate of 1.2e18 means that each user will receive 1.2e18 USDT per 1e18 shares.

It should be noted that the contract does not function properly for vaults which can have their share value decrease over time. Such vaults are luckily not present at this time within the Ambit ecosystem.

convertToShares and convertToAssets may not return the actual values from depositing/withdrawing. An example is in convertToAssets where the vault has zero shares but already value through donations. In this case, the initial shares will have a value higher than 1:1 due to this residual vault value.

Finally, there are some inconsistencies on when convertToAssets, convertToShares and getExchangeRate return 0 or revert. For example, getExchangeRate is zero if totalShares is 0, while convertToAssets and convertToShares would utilise an exchange rate of 1:1 in this instance. Developers should be extremely careful with blindly calling them and go over all exception cases carefully (e.g. zero values of shares or assets etc.).

Any contracts building on top of Vault.sol should not support tokens with fees on transfer nor rebasing tokens. Any non-standard ERC20 token should be dealt with with extreme caution and ideally wrapped in a simpler alternative. Reentrancy tokens should likewise be avoided.


The functions of Vault.sol appear to be inspired by ERC4626 but it should be noted that the contract and its dependencies is not compliant with this standard.

## 2.42.1 Issues & Recommendations

Issue #133

Share distribution can be manipulated by first depositor

Severity

 HIGH SEVERITY

Description

The first depositor can manipulate the shares of all subsequent depositors under the very rare circumstance where no other users deposit within the first six hours after deployment

A user could therefore deposit 1 WEI of the baseAsset and immediately donate a large amount of baseAsset to the vault which linearly increases the amount of `_totalAssets` within the vault.

After some time, depending on the size of the donation, the amount of `_totalAssets` will be such a high value that results in rounding down of subsequent depositors' shares.

Consider the following scenario:

- 1) Alice deposits 1 WEI of baseAsset to the vault, Alice will receive exactly 1 share
- 2) Alice calls donate with  $10\,000e6 - 1$  WEI
- 3) After 6 hours, the amount of `totalAssets` in the vault are exactly  $10000e6$
- 4) Bob deposits  $19\,000e6$  tokens, therefore, the following share calculation will be done:  
$$19000e6 * 1 / 10000e6 = 1.9$$

Since it rounds down, Bob will only receive 1 share, instead of the 1.9 shares he should have received.

This issue is mitigated within the `DepositorVault` contract due to the implementation of the `minSharesOut` parameter. However, it still exist for the `Custodian/Portfolio` contracts.

---

**Recommendation** Consider either refactoring the vault concept to match with OpenZeppelin’s virtual shares concept, or mint a fixed amount of shares to 0xde4d during the first mint.

It should be noted that the latter option does not completely fix the issue but makes it harder to execute it. Combined with the fact that it is highly unlikely that no one deposits within the first 6 hours, we seem this fix as sufficient.

Do note that for significant code changes in the resolution round, a revalidation fee may apply.

---

**Resolution**



This contract is no longer used by the Custodian.

---



## Severity

 INFORMATIONAL

## Description

Line 41

```
uint256 scalar = 10 ** getUnderlyingAsset().decimals();
```

scalar can be marked as an immutable variable, alongside the raw underlying decimals. This avoids fetching these values every time this function is called, saving on gas as decimals is supposed to be constant. Any token with a mutable decimals value would be grossly unsupported regardless, which means that consistently caching the decimals value throughout the codebase for all business logic would even be a security improvement for such tokens.

Consistently caching the decimals throughout the system also means that there is less reliance on this function never reverting. If this function starts reverting due to a paused token for example, this could hinder a lot of critical functionality (e.g. solvability checks for liquidations!). We therefore strongly urge the client to use an internal representation of this value exclusively throughout the system.

## Recommendation

Consider implementing the gas optimizations mentioned above.

## Resolution

 RESOLVED

The decimals variable has been cached in this contract.

---

## 2.42 Vault/Flashlender

FlashLender implements the flashloan logic which originally resided within the DepositorVault contract. It allows authorized addresses by taking out a flashloan via borrowing and repaying from the DepositorVault contract.

A whitelist feature was added which allows for fee-less borrowing. Non-whitelisted borrowers will have to pay a flashloan fee, which is then donated to the DepositorVault as yield-bearing instrument.

### 2.42.1 Issues & Recommendations

No issues found.





**PALADIN**  
BLOCKCHAIN SECURITY