altitude.

ALTITUDE SMART CONTRACTS SECURITY AUDIT REPORT

OXORIO

JULY 16, 2024

CONTENTS

1 AUDIT OVERVIEW

1.1 PROJECT BRIEF

Title	Description
Client	Altitude Labs
Project name	Altitude
Category	Lending, Asset Management
Website	altitude.fi
Repository	github.com/refi-network
Documentation	docs.altitude.fi
Initial Commit	f8344f402066ca51423c5e32b847c96e11d525e0
Final Commit	f7273a4b13e8bb48fb7e47b78390d2e3cbbb2d41
Re-audited Commit	fd86bf702d02eb6e5cad66c5a3b2cdccc601d670
Platform	L1
Network	Ethereum
Languages	Solidity
Lead Auditor	Alexander Mazaletskiy - am@oxor.io
Project Manager	Viktor Mikhailov - viktor@oxor.io

1.2 PROJECT TIMELINE

The key events and milestones of the project are outlined below.

Date	Event
January 3, 2024	Client approached Oxorio requesting an audit.
March 21, 2024	The audit team commenced work on the project.
April 2, 2024	A call with interim results was conducted between the audit team and the client.
April 30, 2024	Submission of the preliminary report #1.
May 13, 2024	Submission of the preliminary report #2.
May 20, 2024	Submission of the comprehensive report.
June 19, 2024	The audit team commenced work on a re-audit of the project.
June 24, 2024	Submission of the preliminary report after re-audit.
July 16, 2024	Submission of the final report after re-audit.

1.3 AUDITED FILES

The following table contains a list of the audited files. The scc tool was used to count the number of lines and assess complexity of the files.

	File	Lines	Blanks	Comments	Code	Complexity
1	contracts/access/Ingress.sol	415	61	70	**284**	19%
2	contracts/common/ProxyExtension.sol	33	4	8	**21**	10%
3	contracts/common/Roles.sol	14	2	6	**6**	0%
4	contracts/common/VaultOperable.sol	32	5	8	**19**	11%
5	contracts/decision-makers/farm-mode-decision/FarmModeDecisionMaker.sol	315	46	70	**199**	13%
6	contracts/interfaces/internal/access/IIngress.sol	127	29	8	**90**	0%
7	contracts/interfaces/internal/decision-makers/farm-mode-decision/IFarmModeDecisionMaker.sol	75	15	5	**55**	0%
8	contracts/interfaces/internal/flashloan/IFlashLoanStrategy.sol	16	4	5	**7**	0%
9	contracts/interfaces/internal/misc/IBorrowVerifier.sol	22	4	5	**13**	0%
10	contracts/interfaces/internal/misc/incentives/position/IPositionIncentivesController.sol	24	8	5	**11**	0%
11	contracts/interfaces/internal/misc/incentives/rebalance/IRebalanceIncentivesController.sol	34	10	5	**19**	0%
12	contracts/interfaces/internal/misc/vault-operable/IVaultOperable.sol	23	6	5	**12**	0%
13	contracts/interfaces/internal/oracles/IChainlinkPrice.sol	18	4	5	**9**	0%
14	contracts/interfaces/internal/oracles/IPriceSource.sol	17	3	6	**8**	0%
15	contracts/interfaces/internal/strategy/farming/IConvexFarmStrategy.sol	42	9	5	**28**	0%
16	contracts/interfaces/internal/strategy/farming/IFarmBuffer.sol	29	11	4	**14**	0%
17	contracts/interfaces/internal/strategy/farming/IFarmBufferStrategy.sol	27	7	4	**16**	0%
18	contracts/interfaces/internal/strategy/farming/IFarmDropMonitorStrategy.sol	18	6	4	**8**	0%
19	contracts/interfaces/internal/strategy/farming/IFarmStrategy.sol	33	11	4	**18**	0%
20	contracts/interfaces/internal/strategy/IFlashLoanCallback.sol	10	2	4	**4**	0%
21	contracts/interfaces/internal/strategy/lending/IAaveStrategy.sol	25	7	5	**13**	0%
22	contracts/interfaces/internal/strategy/lending/ICompStrategy.sol	17	2	5	**10**	0%
23	contracts/interfaces/internal/strategy/lending/ILenderStrategy.sol	67	23	4	**40**	0%
24	contracts/interfaces/internal/strategy/swap/ISwapStrategy.sol	79	18	4	**57**	0%
25	contracts/interfaces/internal/strategy/swap/ISwapStrategyConfiguration.sol	16	5	1	**10**	0%
26	contracts/interfaces/internal/tokens/IDebtToken.sol	17	4	5	**8**	0%
27	contracts/interfaces/internal/tokens/IInterestToken.sol	72	23	5	**44**	0%
28	contracts/interfaces/internal/tokens/ISupplyToken.sol	19	5	5	**9**	0%
29	contracts/interfaces/internal/tokens/ITokensFactory.sol	15	2	4	**9**	0%
30	contracts/interfaces/internal/vault/extensions/configurable/IConfigurableVault.sol	36	6	5	**25**	0%

	File	Lines	Blanks	Comments	Code	Complexity
31	contracts/interfaces/internal/vault/extensions/farmmode/IFarmModeManager.sol	16	4	5	**7**	0%
32	contracts/interfaces/internal/vault/extensions/farmmode/IFarmModeVault.sol	24	6	4	**14**	0%
33	contracts/interfaces/internal/vault/extensions/groomable/IGroomableManager.sol	32	7	5	**20**	0%
34	contracts/interfaces/internal/vault/extensions/groomable/IGroomableVault.sol	27	8	5	**14**	0%
35	contracts/interfaces/internal/vault/extensions/harvestable/IHarvestableManager.sol	39	9	5	**25**	0%
36	contracts/interfaces/internal/vault/extensions/harvestable/IHarvestableVault.sol	37	9	4	**24**	0%
37	contracts/interfaces/internal/vault/extensions/IVaultExtensions.sol	20	3	4	**13**	0%
38	contracts/interfaces/internal/vault/extensions/liquidatable/ILiquidatableManager.sol	27	5	5	**17**	0%
39	contracts/interfaces/internal/vault/extensions/liquidatable/ILiquidatableVault.sol	24	5	4	**15**	0%
40	contracts/interfaces/internal/vault/extensions/snapshotable/ISnapshotableManager.sol	37	7	4	**26**	0%
41	contracts/interfaces/internal/vault/extensions/snapshotable/ISnapshotableVault.sol	33	7	4	**22**	0%
42	contracts/interfaces/internal/vault/extensions/supply-loss/ISupplyLossManager.sol	31	6	5	**20**	0%
43	contracts/interfaces/internal/vault/extensions/supply-loss/ISupplyLossVault.sol	15	3	4	**8**	0%
44	contracts/interfaces/internal/vault/IInterestVault.sol	12	3	4	**5**	0%
45	contracts/interfaces/internal/vault/IVaultCore.sol	111	15	5	**91**	0%
46	contracts/interfaces/internal/vault/IVaultCoreV1Initializer.sol	38	9	8	**21**	0%
47	contracts/interfaces/internal/vault/IVaultRegistry.sol	198	32	5	**161**	0%
48	contracts/interfaces/internal/vault/IVaultStorage.sol	57	23	5	**29**	0%
49	contracts/libraries/types/CommonTypes.sol	32	4	9	**19**	0%
50	contracts/libraries/types/HarvestTypes.sol	54	6	10	**38**	0%
51	contracts/libraries/types/SupplyLossTypes.sol	28	4	8	**16**	0%
52	contracts/libraries/types/VaultTypes.sol	91	12	17	**62**	0%
53	contracts/libraries/utils/CommitMath.sol	403	50	115	**238**	11%
54	contracts/libraries/utils/FlashLoan.sol	22	2	11	**9**	0%
55	contracts/libraries/utils/HealthFactorCalculator.sol	94	9	30	**55**	5%
56	contracts/libraries/utils/Utils.sol	63	5	19	**39**	23%
57	contracts/misc/BorrowVerifier.sol	59	8	12	**39**	8%
58	contracts/misc/incentives/position/PositionIncentivesController.sol	85	13	23	**49**	10%
59	contracts/misc/incentives/rebalance/RebalanceIncentivesController.sol	137	19	30	**88**	3%
60	contracts/oracles/ChainlinkPrice.sol	193	29	48	**116**	16%
61	contracts/oracles/UniswapV3Twap.sol	169	24	39	**106**	16%
62	contracts/strategies/farming/convex/StrategyGenericPool.sol	461	52	85	**324**	15%
63	contracts/strategies/farming/convex/StrategyMeta3Pool.sol	78	9	15	**54**	4%
64	contracts/strategies/farming/convex/StrategyMetaPool.sol	68	8	15	**45**	4%
65	contracts/strategies/farming/convex/StrategyStable2Pool.sol	71	8	15	**48**	4%

	File	Lines	Blanks	Comments	Code	Complexity
66	contracts/strategies/farming/FarmBuffer.sol	99	18	15	**66**	6%
67	contracts/strategies/farming/FarmBufferStrategy.sol	143	28	20	**95**	5%
68	contracts/strategies/farming/FarmDropMonitorStrategy.sol	122	15	29	**78**	12%
69	contracts/strategies/farming/FarmStrategy.sol	180	39	33	**108**	6%
70	contracts/strategies/flashloan/Aavev2FlashLoanStrategy.sol	128	18	25	**85**	8%
71	contracts/strategies/lending/aave/v3/StrategyAaveV3.sol	220	27	46	**147**	3%
72	contracts/strategies/lending/LenderStrategy.sol	219	39	41	**139**	6%
73	contracts/strategies/swap/CurveV2Strategy.sol	329	32	60	**237**	11%
74	contracts/strategies/swap/SwapStrategy.sol	161	20	46	**95**	4%
75	contracts/strategies/swap/SwapStrategyConfiguration.sol	29	5	11	**13**	0%
76	contracts/strategies/swap/UniswapV3Strategy.sol	267	32	42	**193**	10%
77	contracts/tokens/DebtToken.sol	142	18	24	**100**	2%
78	contracts/tokens/InterestToken.sol	337	52	81	**204**	7%
79	contracts/tokens/SupplyToken.sol	212	25	41	**146**	3%
80	contracts/tokens/TokensFactory.sol	53	8	14	**31**	0%
81	contracts/vaults/v1/base/InterestVault.sol	19	4	1	**14**	7%
82	contracts/vaults/v1/base/VaultStorage.sol	122	35	36	**51**	4%
83	contracts/vaults/v1/ERC20/VaultERC20.sol	37	5	13	**19**	0%
84	contracts/vaults/v1/ETH/VaultETH.sol	39	6	11	**22**	14%
85	contracts/vaults/v1/extensions/configurable/ConfigurableManager.sol	116	17	29	**70**	14%
86	contracts/vaults/v1/extensions/configurable/ConfigurableVault.sol	91	9	10	**72**	0%
87	contracts/vaults/v1/extensions/farmmode/FarmModeManager.sol	98	16	22	**60**	12%
88	contracts/vaults/v1/extensions/farmmode/FarmModeVault.sol	97	15	16	**66**	9%
89	contracts/vaults/v1/extensions/groomable/GroomableManager.sol	270	44	35	**191**	14%
90	contracts/vaults/v1/extensions/groomable/GroomableVault.sol	102	12	16	**74**	1%
91	contracts/vaults/v1/extensions/liquidatable/LiquidatableManager.sol	171	26	22	**123**	7%
92	contracts/vaults/v1/extensions/liquidatable/LiquidatableVault.sol	99	11	20	**68**	3%
93	contracts/vaults/v1/extensions/snapshotable/harvest/HarvestableManager.sol	408	59	81	**268**	13%
94	contracts/vaults/v1/extensions/snapshotable/harvest/HarvestableVault.sol	215	23	36	**156**	6%
95	contracts/vaults/v1/extensions/snapshotable/SnapshotableManager.sol	214	19	44	**151**	9%
96	contracts/vaults/v1/extensions/snapshotable/SnapshotableVault.sol	177	17	34	**126**	3%
97	contracts/vaults/v1/extensions/snapshotable/supply-loss/SupplyLossManager.sol	388	52	93	**243**	9%
98	contracts/vaults/v1/extensions/snapshotable/supply-loss/SupplyLossVault.sol	44	5	11	**28**	0%
99	contracts/vaults/v1/ProxyInitializable.sol	60	11	12	**37**	19%
100	contracts/vaults/v1/VaultCore.sol	572	79	115	**378**	8%
101	contracts/vaults/v1/VaultInitializer.sol	150	24	27	**99**	8%
102	contracts/vaults/v1/VaultRegistry.sol	535	58	100	**377**	1%
	Total	**11238**	**1688**	**2177**	**7373**	**7%**

Lines: The total number of lines in each file. This provides a quick overview of the file size and its contents.

Blanks: The count of blank lines in the file.

Comments: This column shows the number of lines that are comments.

Code: The count of lines that actually contain executable code. This metric is essential for understanding how much of the file is dedicated to operational elements rather than comments or whitespace.

Complexity: This column shows the file complexity per line of code. It is calculated by dividing the file's total complexity (an approximation of cyclomatic complexity that estimates logical depth and decision points like loops and conditional branches) by the number of executable lines of code. A higher value suggests greater complexity per line, indicating areas with concentrated logic.

1.4 PROJECT OVERVIEW

DeFi loans are typically over-collateralized and capital-inefficient, Altitude is a non-custodial protocol that optimizes DeFi loans.
Altitude actively manages users' debt and collateral in real-time, optimizing capital efficiency.
When a user takes out a loan from the Altitude it will be optimized by:

- Continuously refinancing debt at the best available rates.
- Actively managing dormant capital to generate yield.
- Channeling the generated yield towards reducing user debt.

Users can interact with the protocol through the following functions. These functions are run against a specific vault:

- `Deposit` - The deposit function allows a user to transfer the supply asset of the vault and receive supplyTokens in exchange.

```
deposit(uint256 amount, address onBehalfOf)
```

`amount` - the amount the user wants to deposit. The user must approve this amount prior to deposit.
`onBehalfOf` - the address onBehalfOf who this deposit is made, typically the user themselves
Upon receiving a deposit the vault will deploy this collateral into the active lending provider to be used as collateral.

- `Borrow` - User borrows the specified amount of borrow assets from the vault and receives borrowTokens in exchange to represent their debt.

```
borrow(uint256 amount)
```

`amount` - the amount the user wants to borrow
Upon receiving a request for a borrow the vault will check how this request would affect the users position and if within specified margins will borrow the required amount from the active lending provider.

- `Withdraw` - User transfers supplyTokens and receives the vault supply assets in return

```
withdraw(uint256 amount, address to)
```

`amount` - amount the user wants to withdraw
`to` - address to which the funds should be sent
The user can withdraw up a loan-to-value ratio of the specified supplyThreshold.

- `Repay` - User transfers vault borrow assets to reduce their own debt or the debt of another user who has approved them to repay

```
repay(uint256 amount, address onBehalfOf)
```

`amount` - amount the user wants to withdraw
`onBehalfOf` - address onBehalfOf who the loan should be repaid.

- `Claim Rewards` - For users who don't have a loan their yield rewards will accumulate directly to their account in the vault borrow asset. These can be withdrawn through the claimRewards function.

```
claimRewards(uint256 amountRequested)
```

`amountRequested` - amount the user wants to withdraw

There are a number of other functions that can be utilized by users in specific scenarios.

- `depositAndBorrow` - combining deposit and borrow in a single transaction.
- `repayAndWithdraw` - combining repaying and withdrawing in a single transaction.
- `commitUser` - update an individual account, recognising earnings/losses from yield farming.
- `liquidateUsers` - liquidate one or more user unhealthy user positions.
- `allowOnBehalf` - allowing a users to authorize a specific address to act on its behalf
- `borrowOnBehalfOf` - a pre-approved user borrows assets on behalf of another user
- `repayBadDebt` - repaying bad debt for a user who's supply token balance is 0.

1.4.1 Researched Attack Vectors

During the analysis of the Altitude protocol, potential attack vectors were identified. Appropriate measures were taken to verify and eliminate these vectors:

1. **Oracle Attacks and Price Manipulation** - Attackers employ various tactics, such as manipulating the underlying asset price or the oracle's feed, to artificially increase vault token values. They also utilize flash loans to borrow significant capital, manipulate conversion rates, and repay loans before price corrections take effect.

2. **Sandwich Attacks During Swaps** - This attack exploits market dynamics by surrounding pending transactions with two malicious transactions, manipulating asset prices to the advantage of the attacker.
3. **Reentrancy with Hookable Contract** - Contracts are susceptible to these attacks when a malicious contract repeatedly calls back into the vulnerable contract before the original execution is completed. This can lead to unauthorized withdrawals or manipulations of the protocol's state.
4. **Cross Contract and View Function Reentrancy** - This problem involves vulnerabilities where the behavior of a smart contract depends on the state of another contract, also when interacting with view functions. Attacker can exploit outdated state information obtained from view functions to carry out reentrancy attacks, manipulating the logic of the protocol.
5. **Access Control Attacks** - Manipulating users or administrators through social engineering attacks, such as phishing, can result in unauthorized access to sensitive information or actions. The lack of proper implementation of access control at the contract level, coupled with insufficient role separation, heightens the risk of malicious actors gaining unwarranted privileges or executing unauthorized transactions within the system.
6. **Mathematics** - The issue arises due to imprecise handling of fractions in contracts used in computations. Especially in protocols that involve significant mathematical operations, leading to rounding errors and precision loss.

1.5 CODEBASE QUALITY ASSESSMENT

The Codebase Quality Assessment table offers a comprehensive assessment of various code metrics, as evaluated by our team during the audit, to gauge the overall quality and maturity of the project's codebase. By evaluating factors such as complexity, documentation and testing coverage to best practices, this table highlights areas where the project excels and identifies potential improvement opportunities. Each metric receives an individual rating, offering a clear snapshot of the project's current state, guiding prioritization for refactoring efforts, and providing insights into its maintainability, security, and scalability. For a detailed description of the categories and ratings, see the Codebase Quality Assessment Reference section.

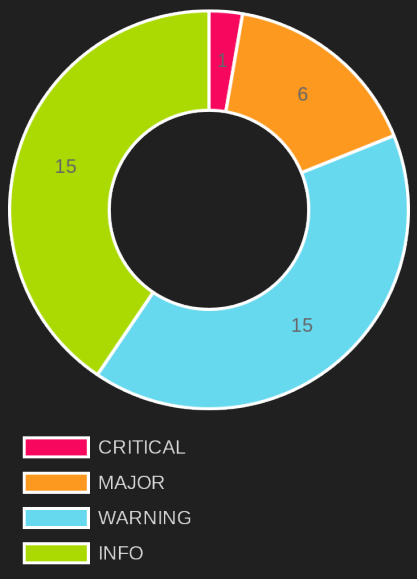
Category	Assessment	Result
Access Control	The project's codebase implements a robust access control mechanism with multiple differentiated roles to manage system functionalities efficiently. Additionally, it includes validations to filter out prohibited addresses under sanctions and permits authorized addresses.	**Good**
Arithmetic	The project diligently manages arithmetic operations to ensure accuracy and security.	**Good**
Complexity	The project benefits from a well-structured modular architecture that enhances readability and maintainability. However, the complexity introduced by the upgradeable scheme using proxy extensions warrants careful consideration.	**Good**
Data Validation	The project performs data validation across many components, but a significant portion of the issues highlighted in this report stem from insufficient validation processes. It is crucial to enhance the validation mechanisms to address these deficiencies and improve the overall robustness of the system.	**Fair**
Decentralization	The project does not incorporate a decentralized approach to management, and therefore, the metric is not applicable in this context.	**Not Applicable**
Documentation	The project's documentation effectively explains the complex logic integral to the system. However, it lacks comprehensive details on the architecture and the interactions among contracts. This absence of a detailed architectural blueprint could impede understanding of the overall system design and operational coherence.	**Good**

Category	Assessment	Result
External Dependencies	The project effectively manages a significant number of external dependencies, including integrations with prominent projects such as `Compound`, `Convex`, `Curve`, `Aave`, and `Uniswap V3`. While some of these integrations were outside the scope of this audit, those that were reviewed exhibited robust implementation practices.	**Excellent**
Error Handling	The project demonstrates competent exception handling throughout the codebase. However, it is important to address the issues outlined in the report that highlight potential error scenarios, including several instances where necessary `revert` statements are missing.	**Good**
Logging and Monitoring	The project exhibits excellent logging capabilities, recording all important events within the system. This comprehensive logging framework enables the effective use of third-party monitoring services such as `Tenderly` or `Forta`, which facilitate real-time data analysis and enhance the ability to track system performance and security incidents accurately.	**Excellent**
Low-Level Calls	The project is free from low-level calls, ensuring a higher level of security by avoiding potential pitfalls associated with direct, low-level interactions with the blockchain.	**Not Applicable**
Testing and Verification	The codebase exhibits commendable test coverage, demonstrating a strong commitment to verifying functionality and reliability.	**Good**

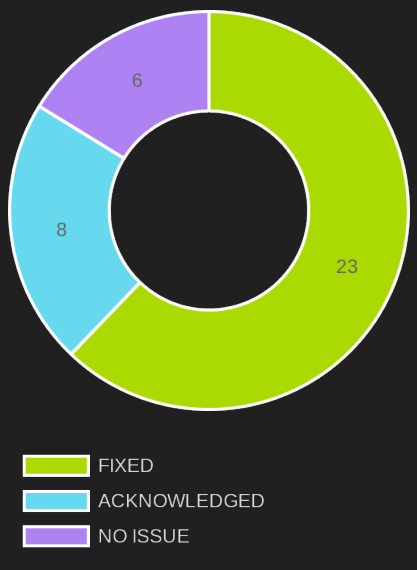
1.6 SUMMARY OF FINDINGS

The table below provides a comprehensive summary of the audit findings, categorizing each by status and severity level. For a detailed description of the severity levels and statuses of findings, see the Findings Classification Reference section.

Severity	TOTAL	NEW	FIXED	ACKNOWLEDGED	NO ISSUE
CRITICAL	**1**	0	1	0	0
MAJOR	**6**	0	3	1	2
WARNING	**15**	0	8	5	2
INFO	**15**	0	11	2	2
TOTAL	**37**	**0**	**23**	**8**	**6**



Issue distribution by severity



Issue distribution by status

This table provides an overview of the findings across the audited files, categorized by severity level. The table enables to quickly identify areas that require immediate attention and prioritize remediation efforts accordingly.

File	TOTAL	CRITICAL	MAJOR	WARNING	INFO
contracts/vaults/v1/extensions/snapshotable/harvest/HarvestableManager.sol	**8**	0	2	1	5
contracts/tokens/InterestToken.sol	**5**	1	1	1	2
contracts/decision-makers/farm-mode-decision/FarmModeDecisionMaker.sol	**4**	0	0	3	1
contracts/vaults/v1/VaultCore.sol	**4**	0	0	2	2
contracts/vaults/v1/VaultRegistry.sol	**3**	0	0	3	0
contracts/vaults/v1/extensions/liquidatable/LiquidatableManager.sol	**3**	0	1	0	2
contracts/vaults/v1/extensions/snapshotable/harvest/HarvestableVault.sol	**3**	0	1	0	2
contracts/access/Ingress.sol	**2**	0	0	2	0
contracts/libraries/utils/CommitMath.sol	**2**	0	0	0	2
contracts/strategies/farming/convex/StrategyGenericPool.sol	**2**	0	0	2	0
contracts/vaults/v1/extensions/groomable/GroomableManager.sol	**2**	0	1	0	1
contracts/common/VaultOperable.sol	**1**	0	0	1	0
contracts/libraries/utils/HealthFactorCalculator.sol	**1**	0	0	0	1
contracts/misc/incentives/rebalance/RebalanceIncentivesController.sol	**1**	0	0	0	1
contracts/strategies/farming/FarmBuffer.sol	**1**	0	0	1	0
contracts/strategies/farming/FarmBufferStrategy.sol	**1**	0	0	0	1
contracts/tokens/DebtToken.sol	**1**	0	0	1	0
contracts/tokens/TokensFactory.sol	**1**	0	0	1	0
contracts/vaults/v1/ERC20/VaultERC20.sol	**1**	0	0	1	0
contracts/vaults/v1/VaultInitializer.sol	**1**	0	0	0	1
contracts/vaults/v1/extensions/configurable/ConfigurableManager.sol	**1**	0	0	0	1
contracts/vaults/v1/extensions/farmmode/FarmModeVault.sol	**1**	0	0	1	0
contracts/vaults/v1/extensions/groomable/GroomableVault.sol	**1**	0	0	0	1
contracts/vaults/v1/extensions/liquidatable/LiquidatableVault.sol	**1**	0	0	0	1
contracts/vaults/v1/extensions/snapshotable/SnapshotableManager.sol	**1**	0	0	0	1
contracts/vaults/v1/extensions/snapshotable/SnapshotableVault.sol	**1**	0	0	0	1

1.7 CONCLUSION

Overall, the project's codebase demonstrates commendable quality. During the initial review, some areas of concern were identified, particularly in arithmetic operations, insufficient data validation, and the handling of edge cases. A thorough review and resolution of these issues were conducted to enhance the system's robustness and security. By addressing these identified problems, the project has significantly improved its resilience against potential vulnerabilities and ensured a more reliable operational framework. We also recommend fuzz testing and additional audits to increase the level of reliability.

2 FINDINGS REPORT

OXORIO

2.1 CRITICAL

C-01	Withdrawal without considering loan interest creates bad debt in `InterestToken`
Severity	**CRITICAL**
Status	• FIXED

Location

File	Location	Line
InterestToken.sol	contract `InterestToken` > function `calcIndex`	230

Description

In the function `calcIndex` of contract `InterestToken`, there is a possibility to execute the function in such a way that it sets the `interestIndex` to zero for `DebtToken` and does not account for the interest on the debt in the protocol.

In such a scenario, it becomes possible to repay debts without considering accrued interest and withdraw deposits, leaving a loss for the remaining users in the protocol.

Let's break down the attack into several steps.

`Step 1`. First, the hacker will need to repay the debt in the lender directly, bypassing the Altitude protocol. For example, for Compound, this would be calling the function `repayBorrowBehalf` and paying off the debt of the `StrategyCompoundV2` contract.

Similar functions for repaying debt on behalf of another user are also available in CompoundV3, AAVEv2 and AAVEv3.

`Step 2`. Next, any function call in Altitude will first update the `interestIndex` of the `DebtToken`, invoking the `calcIndex` function (`SnapshotableVaultV1.updatePosition` -> `_updateInterest` -> `debtToken.snapshot` -> `calcNewIndex` -> `calcIndex`) with `balanceNew=0`. The new balance will be zero since the hacker repaid the entire debt. Inside the function, this will lead to a call to `_calcIndexDecrease`:

```
uint256 indexDecrease = Utils.divRoundingUp(
    interestIndex_ * (balancePrev - balanceNew),
    balancePrev
);

return interestIndex_ - indexDecrease;
```

Ultimately, the `interestIndex` variable will be set to `0` and will not be able to take another value in the `calcIndex` function because the `_calcIndexDecrease` and `_calcIndexIncrease` functions will always return `0` if the first parameter is passed as zero.

`Step 3`. Now, when `interestIndex` is `0`, calling `debtToken.balanceOf(account)` for any user will return the debt amount without indexing, as in the `calcBalanceAtIndex` function, when the previous index is zero, the balance remains unchanged:

```
function calcBalanceAtIndex(
    uint256 balance,
    uint256 fromIndex,
    uint256 toIndex
) internal pure returns (uint256 balanceAtIndex) {
// ...
    balanceAtIndex = balance;
    if (fromIndex > 0) {
        balanceAtIndex = divRoundingUp(balance * toIndex, fromIndex);
    }
}
```

However, interest will continue to accrue on the Altitude debt in Compound.

`Step 4`. The hacker can then borrow the debt through Altitude and repay it after some time, without paying interest on the funds. This is possible if another user keeps a deposit in the protocol, covering the accrued losses.

A test describing this case:

```
it.only("Cancel debt indexation and leave bad debt in the protocol", async function () {
  let cUSDC: ICErc20 = await ethers.getContractAt(
    "ICErc20",
    utils.cUSDCAddress
  );

  const alice = signers[1];
  const bob = signers[2];
```

```
  await ingressControl
    .connect(alice)
    .setDepositLimits(
      ethers.utils.parseEther("0"),
      ethers.utils.parseEther("1000000000"),
      ethers.utils.parseEther("1000000000")
    );

  // Create deposits
  const aliceDepositAmount = ethers.utils.parseEther("10000");
  const bobDepositAmount = ethers.utils.parseEther("10");
  await vaultEth.connect(alice).deposit(aliceDepositAmount, alice.address, {
    value: aliceDepositAmount,
  });
  await vaultEth.connect(bob).deposit(bobDepositAmount, bob.address, {
    value: bobDepositAmount,
  });

  // Alice borrows 1 USDC through Altitude and
  // repay it directly through Compound on behalf of the Compound strategy contract
  const debtAmountInitial = ethers.utils.parseUnits("1", 6);
  await vaultEth.connect(alice).borrow(debtAmountInitial);
  await usdc.connect(alice).approve(cUSDC.address, debtAmountInitial);
  await cUSDC
    .connect(alice)
    .repayBorrowBehalf(compoundStrategy.address, debtAmountInitial);

  // Update the position to see the zero interestIndex of debtToken
  await vaultEth.connect(alice).updatePosition(alice.address);
  expect(await debtToken.interestIndex()).to.be.equal(0);

  // Alice borrows 14000000 USDC
  const aliceDebtAmount = ethers.utils.parseUnits("14000000", 6);
  await vaultEth.connect(alice).borrow(aliceDebtAmount);

  // Wait for 7200 blocks(~24 hours)
  await network.provider.send("hardhat_mine", ["0x1c20"]);

  // Let's check the difference between the value of Alice's debt in the protocol - 14kk
USDC,
  // and the Altitude's entire debt to Compound.
  // As you can see, ~1325 USDC has accumulated, which is not taken into account for Alice.
  await vaultEth.connect(alice).updatePosition(alice.address);
  expect(await debtToken.balanceOf(alice.address)).to.be.equal(14000000000000);
```

```
  expect(await debtToken.totalSupply()).to.be.equal(14001325071306);
  expect(await debtToken.interestIndex()).to.be.equal(0);

  // Alice repays all of her original debt of 14kk USDC without paying anything in interest
  await usdc.connect(alice).approve(vaultEth.address, aliceDebtAmount);
  await vaultEth.connect(alice).repay(aliceDebtAmount, alice.address);

  // Alice withdraws her initial deposit, plus accrued interest.
  const aliceDepositAmountWithAccruedInterest = await supplyToken.balanceOf(
    alice.address
  );
  await vaultEth
    .connect(alice)
    .withdraw(aliceDepositAmountWithAccruedInterest, alice.address);

  // The bottom line is that Alice withdrew her funds,
  // while earning money on the deposit, and did not pay for using the loan

  // With all this, Bob, who did not borrow funds, now cannot withdraw his initial deposit,
  // since the protocol still owes Compound interest on Alice's debt.
  await expect(
    vaultEth.connect(bob).withdraw(bobDepositAmount, bob.address)
  ).to.be.revertedWith("VC_V1_UNHEALTHY_VAULT_RISK");

  // Although the balanceOf function for all users returns a zero balance on the debt within
Altitude
  expect(await debtToken.balanceOf(alice.address)).to.be.equal(0);
  expect(await debtToken.balanceOf(bob.address)).to.be.equal(0);
  expect(await debtToken.balanceOf(vaultEth.address)).to.be.equal(0);
});
```

Recommendation

We recommend reconsidering the conditions for updating the debt balance and accounting for accumulated interest in the protocol in the case of a zero index, in order to avoid the situation of a discrepancy between the stored debt in Altitude and the actual debt in the lender.

Update

Initial fix in commit [f440d61a298d0181147aca2ce8d842af04278946](), final fix in commit [c62937f535df35527aff54d2bf80513445399362]().

Altitude's response

Fixed in commit f440d61a298d0181147aca2ce8d842af04278946.

Oxorio's response

In the function `mint` of contract `InterestToken`, it is possible to set an initial `interestIndex`, which depends on the size of the `amount`:

```
if (interestIndex == MATH_UNITS) {
    uint256 interestIndex_ = interestIndex;
    uint256 indexIncrease = interestIndex_ * amount;

    interestIndex += indexIncrease;
}
```

In this case, if a user with `interestIndex == MATH_UNITS` takes a very large loan and immediately repays it using `borrow/repay`, the `interestIndex` will be set very high and will continue to be used as such.

Even a single token with `decimal = 18` as `amount` will yield a value of `1*10^18 * 10^20 = 10^38` when multiplied by `MATH_UNITS`.

Subsequently, such an `interestIndex` will need to be multiplied by the token balance - `10^38 * N*10^18 = N*10^56`. Therefore, if a large index value is initially set and then used with large token balances, it can result in an overflow during subsequent balance indexing.

Altitude's response

Fixed in commit c62937f535df35527aff54d2bf80513445399362.

Oxorio's response

We confirm that commit c62937f535df35527aff54d2bf80513445399362 is a valid fix.

2.2 MAJOR

M-01	Possible overflow in `HarvestableManager`
Severity	**MAJOR**
Status	• NO ISSUE

Location

File	Location	Line
HarvestableManager.sol	contract `HarvestableManager` > function `withdrawReserve`	294
HarvestableManager.sol	contract `HarvestableManager` > function `withdrawReserve`	303

Description

In the function `withdrawReserve` of contract `HarvestableManager` there is possible overflow.

```
uint256 maxAmount = IERC20(borrowUnderlying).balanceOf(address(this)) +
harvestStorage.vaultReserve - farmModeStorage.farmModeReserve;
uint256 readyAmount = maxAmount - harvestStorage.vaultReserve;
```

The `maxAmount` calculation does not take into account the possible case that `IERC20(borrowUnderlying).balanceOf(address(this)) + harvestStorage.vaultReserve` may be less than `farmModeStorage.farmModeReserve`.

At the same time, the calculation of `readyAmount` does not take into account the possibility that the `maxAmount` may be less than `harvestStorage.vaultReserve`.

Recommendation

We recommend reviewing the logic and adding conditions for calculating values.

Update

Altitude's response

Due to how farmModeReserve is set this is not an issue. However for readability we have clarifying this part of the code-base.

Oxorio's response

In our opinion, these changes look like FIXED, since conditions were added to the code that solve the original issue, and which would not be needed otherwise.

Altitude's response

To clarify, for the above to be an issue we would need to be able to create a case where:

```
IERC20(borrowUnderlying).balanceOf(address(this)) < farmModeStorage.farmModeReserve
```

Focussing on this, we need to look at all cases where farmModeReserve can be increased. The only place where this happens is in the `FarmmModeManager.sol` contract, specifically in the `disableFarmMode()` function.

- `farmModeStorage.farmModeReserve += farmAmount - debtToRepay;` represents the amount the vault has deployed in the farm, minus any associated borrowing costs
- `farmModeStorage.farmModeReserve += lenderRewards;` represents the amount we have recognised in lender rewards

Here the `farmAmount` is set by checking the difference in erc20 balance before and after withdrawing from the farm. `lenderRewards` is set by identifying exactly how many rewards are recognised from the lender. Based on this these cases in themselves can never lead to `IERC20(borrowUnderlying).balanceOf(address(this)) < farmModeStorage.farmModeReserve` being an issue.

The other option then would be for `IERC20(borrowUnderlying).balanceOf(address(this))` to reduce without `farmModeStorage.farmModeReserve` or `harvestStorage.vaultReserve` being updated. We don't see how this would be possible.

As such we do agree the code could have been more readable (which is why we made the updates), but wouldn't consider that a major issue.

M-02	Zero debt size sets `interestIndex` to `0` permanently in `InterestToken`
Severity	**MAJOR**
Status	• FIXED

Location

File	Location	Line
InterestToken.sol	contract `InterestToken` > function `calcIndex`	230

Description

In the function `calcIndex` of contract `InterestToken`, the condition for decreasing the index works when the size of debt decreases for `DebtToken`:

```
uint256 interestIndex_ = interestIndex;
uint256 balanceNew = totalSupply();

if (
    balanceOld > balanceNew &&
    !ILenderStrategy(activeLenderStrategy).hasSupplyLoss()
) {
    interestIndex_ = _calcIndexDecrease(
        interestIndex_,
        balanceOld,
        balanceNew
    );
}
```

However, if `balanceNew` becomes `0`, then `interestIndex_` will drop to `0` in the `_calcIndexDecrease` function:

```
uint256 indexDecrease = Utils.divRoundingUp(
    interestIndex_ * (balancePrev - balanceNew),
    balancePrev
);
```

```
return interestIndex_ - indexDecrease;
```

In such a case, the `interestIndex` variable will be set to `0` and will not be able to take another value in the `calcIndex` function because the `_calcIndexDecrease` and `_calcIndexIncrease` functions will always return `0` if the first parameter is passed as zero.

It is worth noting that in the `InterestToken` contract, there is a setter `setInterestIndex`, which sets `interestIndex` directly, but for `DebtToken`, it is called only once and takes the current `interestIndex`, which, in the case of `0`, means setting it back to `0`.

Recommendation

We recommend reviewing the conditions of the interestIndex calculation to avoid a situation in which it could become equal to zero and get stuck at the zero level, as a result of which interest on the debt will no longer be taken into account in the protocol.

Update

Final fix in commit [f440d61a298d0181147aca2ce8d842af04278946](f440d61a298d0181147aca2ce8d842af04278946).

Altitude's response

Fixed together with C-01 as related

Oxorio's response

We confirm that commit [f440d61a298d0181147aca2ce8d842af04278946](f440d61a298d0181147aca2ce8d842af04278946) is a valid fix.

M-03	Excessive debt repayment locks liquidation process in `LiquidatableManager`
Severity	**MAJOR**
Status	• FIXED

Location

File	Location	Line
LiquidatableManager.sol	contract `LiquidatableManager` > function `liquidateUsers`	137

Description

In the function `liquidateUsers` of contract `LiquidatableManager`, a revert occurs when attempting to repay more debt than Altitude owes in total.

Such a situation is possible when working with Compound as a lender. It is necessary to obtain a situation where the `debtToken` balance is greater than the debt Altitude owes to Compound.

This can be achieved, for example, by initially repaying the entire Altitude debt in Compound directly. Then, upon the next index update, the `interestIndex` for `DebtToken` will be set to `0`, and the balances of all users in `DebtToken` will not be updated relative to Compound.

Subsequently, if a user borrows through Altitude and repays a portion of this debt directly in Compound, bypassing Altitude, the actual debt in Compound will be less than the one accounted for in Altitude.

If we consider a scenario where there is only one user in Altitude, and their position falls under liquidation, the liquidator will attempt to repay a larger amount to Compound than is actually required.

Such behavior in Compound will result in a revert when calling the `repayBorrow` function at the moment of deduction from the entire debt, and consequently, the position will not be liquidated:

```
/*

 * We calculate the new borrower and total borrow balances, failing on underflow:
 *  accountBorrowsNew = accountBorrows - repayAmount
 *  totalBorrowsNew = totalBorrows - repayAmount
```

```
 */
(vars.mathErr, vars.accountBorrowsNew) = subUInt(vars.accountBorrows, vars.repayAmount);
if (vars.mathErr != MathError.NO_ERROR) {
    return failOpaque(Error.MATH_ERROR,
FailureInfo.REPAY_BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED, uint(vars.mathErr));
}
```

Thus, the user is able to prevent the liquidation of their position by making small repayments directly in Compound. Furthermore, such a lock on the liquidation of a user's position in Altitude may lead to the liquidation of Altitude's position in the lender, for example, in the event of a rapid decline in the collateral price.

On the other hand, when working with AAVE as a lender, there will be no revert during repayment, and the debt will be repaid. However, the "excess" will not be spent and will remain in the AAVE strategy contract. Moreover, these excess funds cannot be withdrawn from the strategy contract since it does not provide for the withdrawal of accidentally deposited tokens.

Test for the Compound case:

```
it("Liquidation process reset", async function () {
  let cUSDC: ICErc20 = await ethers.getContractAt(
    "ICErc20",
    utils.cUSDCAddress
  );

  await ingressControl
    .connect(signers[1])
    .setDepositLimits(
      ethers.utils.parseEther("0"),
      ethers.utils.parseEther("1000000000"),
      ethers.utils.parseEther("1000000000")
    );

  // Create deposit of 10 ETH
  const depositAmount = ethers.utils.parseEther("10");
  await vaultEth
    .connect(signers[1])
    .deposit(depositAmount, signers[1].address, {
      value: depositAmount,
    });

  // Borrow 1 USDC through Altitude and repay it directly through Compound
  const debtAmountInitial = ethers.utils.parseUnits("1", 6);
  await vaultEth.connect(signers[1]).borrow(debtAmountInitial);
```

```
await usdc.connect(signers[1]).approve(cUSDC.address, debtAmountInitial);
await cUSDC
  .connect(signers[1])
  .repayBorrowBehalf(compound.address, debtAmountInitial);

// interestIndex of debtToken is zero
await vaultEth.connect(signers[1]).updatePosition(signers[1].address);
expect(await debtToken.interestIndex()).to.be.equal(0);

// Borrow 13000 USDC and repay 7000 USDC through Compound
const debtAmount = ethers.utils.parseUnits("13000", 6);
await vaultEth.connect(signers[1]).borrow(debtAmount);

const repayAmount = ethers.utils.parseUnits("7000", 6);
await usdc.connect(signers[1]).approve(cUSDC.address, repayAmount);
await cUSDC
  .connect(signers[1])
  .repayBorrowBehalf(compound.address, repayAmount);

// Now Altitude has a debt of ~6000 USDC in Compound,
// but continues to consider debt as 13000 USDC in Altitude in debtTokens
expect(await debtToken.balanceOf(signers[1].address)).to.be.equal(
  13000000000
);
expect(
  await cUSDC.callStatic.borrowBalanceCurrent(compound.address)
).to.be.equal(6000000327);
expect(await debtToken.interestIndex()).to.be.equal(0);

// Position is not liquidatable
expect(await vaultEth.isUserForLiquidation(signers[1].address)).to.be.equal(
  false
);

// Change ETH price from 2000 to 1000 USDC/ETH to decrease collateral value
compoundPriceMock.getUnderlyingPrice
  .whenCalledWith(utils.cEtherAddress)
  .returns(ethers.utils.parseUnits("1000", 36 - 18));

// Position is liquidatable now
expect(await vaultEth.isUserForLiquidation(signers[1].address)).to.be.equal(
  true
);

// The liquidator is trying to repay the amount of debt(13000 USDC)
```

```
  // In fact, only 6500 USDC will be attempted to be repaid due to
maxPositionLiquidation==50%.
  await claimToken(signers[0], usdc.address, debtAmount);
  await usdc.connect(signers[0]).approve(vaultEth.address, debtAmount);
  // But the repayment will fail
  // because Compound has fewer borrowed tokens (6000 USDC) for Altitude
  // than the liquidator is trying to repay (6500 USDC).
  await expect(
    vaultEth
      .connect(signers[0])
      .liquidateUsers([signers[1].address], debtAmount)
  ).to.be.revertedWith("SC_BASE_REPAY_FAILED");

  // Reduce the maxPositionLiquidation variable to 40%.
  // Now the liquidator can liquidate 5200 USDC from 13000 USDC of debt(5200=13000*0.4)
  const liquidatableManagerAddress = (
    await vaultEth.connect(signers[0]).getLiquidationConfig()
  )[0];
  await vaultEth.connect(signers[1]).setLiquidationConfig({
    liquidatableManager: liquidatableManagerAddress,
    maxPositionLiquidation: ethers.utils.parseUnits("0.4", 18),
    liquidationBonus: LIQUIDATION_BONUS,
    minUsersToLiquidate: MIN_USERS_TO_LIQUIDATE,
    minRepayAmount: MIN_REPAY_AMOUNT,
  });

  // Liquidation passed because 5200 USDC < ~6000 USDC of borrowed tokens in Compound
  await vaultEth
    .connect(signers[0])
    .liquidateUsers([signers[1].address], debtAmount);

  expect(await debtToken.balanceOf(signers[1].address)).to.be.equal(
    13000000000 - 5200000000
  );
  expect(
    await cUSDC.callStatic.borrowBalanceCurrent(compound.address)
  ).to.be.equal(6000000704 - 5200000000);
  expect(await debtToken.interestIndex()).to.be.equal(0);
});
```

Recommendation

We recommend adding a check to ensure the correspondence between the repaid debt and the actual debt in the lender during the liquidation process to avoid undesirable behavior.

Update

Fixed in commit [f440d61a298d0181147aca2ce8d842af04278946](f440d61a298d0181147aca2ce8d842af04278946).

M-04	Harvest profit deprivation due to resetting `harvestJoiningBlock` in `HarvestableVaultV1`
Severity	**MAJOR**
Status	• ACKNOWLEDGED

Location

File	Location	Line
HarvestableVault.sol	contract `HarvestableVaultV1` > function `_updateEarningsRatio`	210

Description

In the function `_updateEarningsRatio` of contract `HarvestableVaultV1`, the `harvestJoiningBlock` variable is set to the current block number, and this function is called on deposit, repay, position liquidation, and `supplyToken` transfer. Subsequently, the `harvestJoiningBlock` variable is used in calculating the user's share of profits going to the `vaultReserve` - the greater the difference between `harvestJoiningBlock` and the block at the time of harvest, the less profit goes to `vaultReserve`, leaving more for the user. Notably, this calculation is independent of the number of deposits made:

```
if (userHarvestChange >= 0) {
    // Calculate the user's participation ratio in the harvest
    // userHarvestRatio = participatingBlocks / totalHarvestBlocks
    userHarvestRatio = int256(
        (1e18 *
            (harvestSnapshot.blockNumber -
                commit.userHarvestJoiningBlock)) /
            (harvestSnapshot.blockNumber - commit.blockNumber)
    );
}

// Apply the userHarvestRatio to the userHarvestChange
int256 userHarvestChangeNew = (userHarvestRatio *
    userHarvestChange) / 1e18;

// Disincentivise users from only participating briefly in harvests
// Divert earnings for the period the user wasn't fully in the harvest
commit.vaultReserveUncommitted =
```

```
    uint256(userHarvestChange - userHarvestChangeNew) +
    commit.vaultReserveUncommitted;
```

This situation can result in a user receiving no profit from the harvest. Suppose a user made a large deposit a long time ago and then made another deposit just before the harvest, but for a negligible amount. In that case, the `harvestJoiningBlock` updates, and the difference between `harvestJoiningBlock` and the harvest block is minimal - all of the user's harvest profit goes to the `vaultReserve`.

Additionally, it is worth noting that functions (`deposit`, `repay`, `SupplyToken.transfer`) that reset `harvestJoiningBlock` can be called on behalf of another user. This would however require either the account to be pre-approved by the user or the current protection to be removed from the function.

For instance, a user can deposit `1` wei for another user just before the harvest, thus depriving them of their earnings. This scenario is possible if one of the functions that reset `harvestJoiningBlock` on behalf of a third party, such as the `deposit` function, is added to the `onBehalfFunctions` mapping. Then, the check in the `onlyAllowedOnBehalf` modifier of the `VaultCoreV1` contract will pass successfully:

```
if (
    allower != allowee &&
    !allowOnBehalfList[allower][allowee] &&
    !onBehalfFunctions[selector]
) {
    revert VC_V1_NOT_ALLOWED_TO_ACT_ON_BEHALF();
}
```

The `onBehalfFunctions` mapping works only for the `onlyAllowedOnBehalf` modifier, and `onlyAllowedOnBehalf` is checked only when functions resetting `harvestJoiningBlock` - `deposit`, `repay`, `SupplyToken.transfer` - are called. This means that any inclusion of the `onBehalfFunctions` mapping in operation will allow one user to deprive other protocol users of their earnings.

A test case describing the above flow:

```
it("Updating the joining block to deprive earnings from harvesting", async function () {
  const alice = signers[1];
  const bob = signers[2];

  await ingressControl
    .connect(alice)
    .setDepositLimits(
```

```
      ethers.utils.parseEther("0"),
      ethers.utils.parseEther("1000000000"),
      ethers.utils.parseEther("1000000000")
    );

  // Create deposits
  const aliceDepositAmount = ethers.utils.parseEther("1000");
  await vaultEth.connect(alice).deposit(aliceDepositAmount, alice.address, {
    value: aliceDepositAmount,
  });

  // harvestJoiningBlock after the first deposit
  const firstJoiningBlock = (await vaultEth.getUserHarvest(alice.address))
    .harvestJoiningBlock;

  // Wait for 7200 blocks(~24 hours)
  await network.provider.send("hardhat_mine", ["0x1c20"]);

  await vaultEth.connect(alice).updatePosition(alice.address);

  // Checking that the harvestJoiningBlock has not changed after 7200 blocks
  expect(
    (await vaultEth.getUserHarvest(alice.address)).harvestJoiningBlock
  ).to.be.equal(firstJoiningBlock);

  // Add the deposit function selector to the onBehalfFunctions mapping
  expect(await vaultEth.onBehalfFunctions("0x6e553f65")).to.be.false;
  await vaultEth.connect(alice).disableOnBehalfValidation(["0x6e553f65"], true);
  expect(await vaultEth.onBehalfFunctions("0x6e553f65")).to.be.true;

  // Now any user can make a deposit for Alice's position.
  // For example, Bob can make a deposit of 1 wei.
  // This action will update the harvestJoiningBlock and
  // thereby deprive Alice of part of the profit during the next harvest.
  await vaultEth
    .connect(bob)
    .deposit(ethers.utils.parseUnits("1", 1), alice.address, {
      value: ethers.utils.parseUnits("1", 1),
    });

  const secondJoiningBlock = (await vaultEth.getUserHarvest(alice.address))
    .harvestJoiningBlock;
  expect(secondJoiningBlock).to.be.equal(firstJoiningBlock.add(7200 + 3));
});
```

Recommendation

We recommend considering the possibility of calculating the share of harvest profits based on the sizes of all deposits and the blocks when they were made, not just the last `harvestJoiningBlock`.

Update

Altitude's response

This is a known limitation. Alternative solutions typically either significantly increase complexity or introduce attack vectors.

We are investigating improvements in this area but it is likely to be in future versions of the protocol. In the meantime this problem will be mitigated as we both limit deposits and can harvest more frequently.

M-05	Absence of whitelist allows injection and distribution of "dirty" cryptocurrency in `HarvestableManager`
Severity	**MAJOR**
Status	• FIXED

Location

File	Location	Line
HarvestableManager.sol	contract `HarvestableManager` > function `injectBorrowAssets`	169

Description

In the function `injectBorrowAssets` of contract `HarvestableManager`, there is no validation through the `Ingress` contract - there is no check for the recipient's presence in the `allowList` and `sanctioned` mappings. Additionally, the function itself is not checked for absence in the `isFunctionDisabled` mapping, which is inherent to other vault functions. Therefore, any user can inject `borrowUnderlying` tokens into the protocol, and these tokens can then be immediately used to repay the user's debt or withdrawn from the protocol as earnings.

This function enables a malicious user to inject "dirty" cryptocurrency into the protocol and distribute it among both their own and other participants. Consequently, he "cleans" a portion of funds obtained through dubious means by blending in with the crowd of protocol users.

All this leads to the possibility of the Altitude protocol itself being banned if suspicions arise of its use for money laundering.

Recommendation

We recommend considering restrictions on the ability to call the `injectBorrowAssets` function by adding whitelists and blacklists of users who can inject `borrowUnderlying` into the protocol.

Update

Fixed in commit 9cd8c2128fa24323529294c58365d9d045f70591.

M-06	`migrationFee` increases borrow without considering `availableBorrow` in `GroomableManager`
Severity	**MAJOR**
Status	• NO ISSUE

Location

File	Location	Line
GroomableManager.sol	contract `GroomableManager` > function `flashLoanCallback`	121

Description

In the function `flashLoanCallback` of contract `GroomableManager`, a flashloan taken for migration to another strategy is repaid by taking a loan through the `borrow` function for the amount of the previous strategy's debt repayment, including the `migrationFee`. However, there is no check for the health factor or the available amount for the loan (`availableBorrow`). This can lead to exceeding the current target threshold, resulting in system imbalance and uncontrolled consequences.

Recommendation

We recommend adding a check to ensure that the amount needed to repay the flashloan is within the available borrow limit.

Update

Altitude's response

We don't think this is an issue, there are a few possible scenarios here:

1. migration fee increases LTV to be less than lender liquidation threshold -> rebalance will be triggered (within same TX)
2. migration fee increases LTV to be more than lender liquidation threshold -> migration will fail

As after each migration we run a `rebalance()` the vault will try to reset itself to its target threshold as part of the migration. In both cases there aren't any new system imbalances that need to be managed.

2.3 WARNING

W-01	Lack of functionality to withdraw stuck tokens
Severity	**WARNING**
Status	• ACKNOWLEDGED

Description

In all contracts of the project, there is no functionality to withdraw stuck tokens. Consequently, when a user mistakenly sends tokens to publicly used project contracts (such as `VaultCoreV1`), these tokens remain stuck without any ability to be withdrawn.

Also, there is a possibility to lock tokens of the protocol itself. For example, if when calling the `repay` function in the `LenderStrategy` contract for the AAVE lender, we pass an `amount` greater than the total debt in AAVE, the remaining funds after repaying the debt will remain on the strategy contract without the ability to withdraw them.

Recommendation

We recommend adding functionality to withdraw stuck tokens from publicly used contracts.

Update

Altitude's response

We have decided not to implement this at this point, we may consider it for future versions.

W-02	Non-zero balance with zero index in `InterestToken`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
InterestToken.sol	contract `InterestToken` > function `calcIndex`	234

Description

In the function `calcIndex` of contract `InterestToken`, it is possible to obtain a zero index with a non-zero balance if the value of `balanceOld` is greater than `interestIndex`, and `balanceNew` approaches zero. This is possible due to rounding up inside the `_calcIndexDecrease` function.

Let's assume we call the `_calcIndexDecrease` function with the following parameters:

- `interestIndex_` = 100
- `balanceOld` = 1000
- `balanceNew` = 9

In this case, due to rounding up in the `divRoundingUp` function, we will have `indexDecrease == interestIndex_`, and as a result, the new index `interestIndex` will be `0`, while the new balance will be greater than zero (`balanceNew = 9`):

```
// numerator = interestIndex_ * (balanceOld - balanceNew)
// denominator = balanceOld

function divRoundingUp(
    uint256 numerator,
    uint256 denominator
) internal pure returns (uint256 result) {
    if (numerator > 0 && denominator > 0) {
        result = numerator / denominator;
        if (result * denominator < numerator) {
            result += 1;
        }
```

```
    }
}
```

Recommendation

We recommend considering possible calculation errors during division and avoiding situations where the index is reset to zero with a non-zero balance.

Update

Fixed in commit [f440d61a298d0181147aca2ce8d842af04278946](f440d61a298d0181147aca2ce8d842af04278946).

W-03	Reference price is set up externally in `StrategyGenericPool`
Severity	**WARNING**
Status	• NO ISSUE

Location

File	Location	Line
StrategyGenericPool.sol	contract `StrategyGenericPool` > function `setReferencePrice`	106

Description

In the function `setReferencePrice` of the contract `StrategyGenericPool`, the reference price of LP tokens is set externally. This value is used in the calculation of the amount of LP and underlying during deposits and withdrawals.

```
function calcLPExpected(
    uint256 inputAmount_
) internal view returns (uint256 minAmount) {
    // ..
    // referencePrice and LP tokens work with 18 decimals
    minAmount =
        Utils.scaleAmount(inputAmount_, farmAssetDecimals, 36) /
        referencePrice;

    minAmount = minAmount - ((minAmount * slippage) / SLIPPAGE_BASE);
}
// ..
function calcUnderlyingExpected(
    uint256 inputAmount_
) internal view returns (uint256 minAmount) {
    // ..

    // referencePrice and LP tokens work with 18 decimals
    minAmount = Utils.scaleAmount(
        inputAmount_ * referencePrice,
        36,
        farmAssetDecimals
    );
```

```
    minAmount = minAmount - ((minAmount * slippage) / SLIPPAGE_BASE);
}
```

As the price of LP tokens depends on the content of the pool, the fixed value of the reference price cannot be reliable during all market conditions, resulting in imprecise calculations of the received amounts.

Recommendation

We recommend using oracle data together with LP pricing formula to assess the price of the LP tokens.

Update

Altitude's response

We have investigated your recommended approach and reached out to a team that originally did something similar for Curve. This team discontinued this approach as Curve pools are not consistent enough and it was time consuming and error prone. As such we don't believe this approach is viable in this case.

W-04	Possibility of complete withdrawal in case of farm loss in `VaultCoreV1`
Severity	**WARNING**
Status	• ACKNOWLEDGED

Location

File	Location	Line
VaultCore.sol	contract `VaultCoreV1` > function `_validateWithdraw`	531

Description

In the function `_validateWithdraw` of contract `VaultCoreV1`, there is no check on the farm balance status, allowing a withdrawal from the farm even in the presence of losses. This leads to unforeseen farm losses not being socialized.

For instance, suppose there are 100 users who deposited `1` ETH each. Considering a 70% `liquidationThreshold` and the price of ETH being `2000` USDC/ETH, the vault pledged users' deposits for a debt of `140000` USDC and deposited it in the farm:

```
100 ETH * 0.7 * 2000 USDC/ETH = 140000 USDC
```

Suddenly, the farm incurs a loss, losing a third of the deposited tokens, thus reducing the vault balance to `~93333` USDC.

In the very next block, a vigilant user, to avoid losses, initiates a withdrawal of their funds amounting to `1` ETH. As a result, the protocol deducts the user's share from the farm balance, `1 ETH * 0.7 * 2000 USDC/ETH = ~1400 USDC`, allowing the user to exit the protocol without any losses. Meanwhile, the user can front-run any actions by the protocol admins, such as imposing restrictions, setting a pause, disabling `farmMode`, etc.

It is evident that there won't be enough money on the farm to allow all users to withdraw. The remaining users will share the losses among themselves.

A test case illustrating a similar scenario:

```
it("Socialization of farm losses", async function () {
  const alice = signers[1];
  const bob = signers[2];
```

```
  await ingressControl
    .connect(alice)
    .setDepositLimits(
      ethers.utils.parseEther("0"),
      ethers.utils.parseEther("1000000000"),
      ethers.utils.parseEther("1000000000")
    );

  // Create deposits
  const aliceDepositAmount = ethers.utils.parseEther("10");
  const bobDepositAmount = ethers.utils.parseEther("10");
  await vaultEth.connect(alice).deposit(aliceDepositAmount, alice.address, {
    value: aliceDepositAmount,
  });
  await vaultEth.connect(bob).deposit(bobDepositAmount, bob.address, {
    value: bobDepositAmount,
  });

  // We do a rebalance. The vault borrows funds from the lender and deposits them into the
farm
  expect(await farmPool.balance()).to.be.equal(0);
  await vaultEth.connect(alice).rebalance();

  // Alice and Bob deposited 10 ETH each.
  // The price of ETH is 2000 USDC/ETH.
  // The maximum that can be borrowed is equal to the liquidationThreshold of 70%.
  // This means their shares in the farm are 10 ETH * 2000 USDC/ETH * 0.7 = 14000 USDC each.
  const price = ethers.utils.parseUnits("1400", 6); // 2000 USDC/ETH * 0.7
  const aliceShare = aliceDepositAmount
    .mul(price)
    .div(ethers.utils.parseUnits("1"));
  const bobShare = bobDepositAmount
    .mul(price)
    .div(ethers.utils.parseUnits("1"));

  const deviation = ethers.utils.parseUnits("20", 6);

  // The farm balance is equal to the sum of Alice's and Bob's shares(28000 USDC)
  const balanceInitial = await farmPool.balance();
  expect(balanceInitial).to.be.closeTo(aliceShare.add(bobShare), deviation);

  // Suppose there is an unexpected loss on the farm and the vault balance drops by a
third(-9333 USDC)
  const loss = balanceInitial.div(3);
```

```
    await farmPool.setVariable("vault", signers[0].address);
    await farmPool.connect(signers[0]).withdraw(loss);
    await farmPool.setVariable("vault", vaultEth.address);

    // Now the vault balance on the farm is 18666 USDC
    const balanceWithLoss = await farmPool.balance();
    expect(balanceWithLoss).to.be.closeTo(balanceInitial.sub(loss), deviation);

    // Obviously this balance is not enough to cover the added shares of Alice and Bob
    // (14000 + 14000 > 18666)
    expect(aliceShare.add(bobShare)).to.be.gt(balanceWithLoss);

    // But, Alice (or Bob) can withdraw
    // her initial deposit without taking into account the loss
    await vaultEth.connect(alice).withdraw(aliceDepositAmount, alice.address);
    expect(await farmPool.balance()).to.be.closeTo(
      balanceWithLoss.sub(aliceShare),
      deviation
    );

    // And now Bob cannot withdraw all his funds and will have to deal with the loss alone
    await expect(
      vaultEth.connect(bob).withdraw(bobDepositAmount, bob.address)
    ).to.be.revertedWith("VC_V1_FARM_WITHDRAW_INSUFFICIENT");
  });
```

Recommendation

We recommend considering adding a check on the farm balance before deducting user funds and making the distribution of unforeseen farm losses fairer.

Update

Altitude's response

We are preparing a fix for this which will be released together with several other improvements.

W-05	No `_disableInitializers` call in the constructor in `VaultRegistryV1`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
VaultRegistry.sol	contract `VaultRegistryV1`	25

Description

In the contract `VaultRegistryV1` the is no call to the `_disableInitializers` method in the constructor. This call is required to lock the implementation contract from initialization.

Recommendation

We recommend calling the `_disableInitializers` method in the constructor of the `VaultRegistryV1` contract.

Update

Final fix in commit d7d36072ee9049c0584c27cec190402eabbbd668.

Altitude's response

Uninitialised proxy (implementation) vulnerability applies to UUPS class of proxies, ie to such that the upgrade logic resides with the implementation code. This is not what Altitude is using.

Altitude implements upgradeability in a way aligned with the transparent upgradable proxy ie. The upgrade logic resides in the proxy contract itself. In other words there are no delegate calls in the registry that could be abused by the attacker trying to exploit the initialisation function. Therefore disabling the initialisation on the implementation contract will not have any bearing on the security.

Oxorio's response

We agree with your statement, but we recommend using the `_disableInitializers` method as an extra layer of protection, as advised by OpenZeppelin team.

The absence of the `_disableInitializers` method could potentially allow an attacker to gain administrative rights in the implementation contract by calling the `initialize` function. After obtaining such rights, they could use it to perform, for example, phishing attacks.

Altitude's response

Fixed in d7d36072ee9049c0584c27cec190402eabbbd668.

Oxorio's response

We confirm that commit d7d36072ee9049c0584c27cec190402eabbbd668 is a valid fix.

W-06	Incorrect farm mode disable condition in `FarmModeDecisionMaker`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
FarmModeDecisionMaker.sol	contract `FarmModeDecisionMaker` > function `manageFarmMode`	268

Description

In the function `manageFarmMode` of the contract `FarmModeDecisionMaker`, the farm mode is disabled in case of the strict inequality `farmDrop > config.farmDropThresholdMin` returns `true`. According to the documentation provided, the farm mode should be disabled also when `farmDrop = config.farmDropThresholdMin`.

Recommendation

We recommend changing the inequality to `farmDrop >= config.farmDropThresholdMin`.

Update

Altitude's response

Documentation has been updated.

W-07	Insufficient reference price validation in `StrategyGenericPool`
Severity	**WARNING**
Status	• NO ISSUE

Location

File	Location	Line
StrategyGenericPool.sol	contract `StrategyGenericPool` > function `setReferencePrice`	106

Description

In the function `setReferencePrice` of the contract `StrategyGenericPool`, the reference price is not validated to be within meaningful bounds.

Recommendation

We recommend validating the price against the value of the assets in the pool and setting the upper and the lower bound for the price.

Update

Altitude's response

As the pool could move in any direction it seems like letting the admin decide the value is correct, unless we can mathematically guarantee the price would be restricted to certain values.

Oxorio's response

If it is assumed that the administrator can set the correct price under any conditions, then we agree with the NO ISSUE status.

W-08	`DEFAULT_ADMIN_ROLE` is assigned to `msg.sender` during contracts deployment
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
VaultRegistry.sol	contract `VaultRegistryV1` > function `initialize`	67
Ingress.sol	contract `Ingress` > function `constructor`	86
FarmModeDicisionMaker.sol	contract `FarmModeDecisionMaker` > function `constructor`	59

Description

In the contracts `VaultRegistryV1`, `Ingress`, and `FarmModeDecisionMaker`, the `DEFAULT_ADMIN_ROLE` is assigned to `msg.sender`. If these contracts are deployed through a third-party contract or deployer address that is not accessible for management, access to the system's management will be lost.

Recommendation

We recommend adding a parameter in the `initialize` function and constructors of the contracts for a separate address to be assigned the `DEFAULT_ADMIN_ROLE`.

Update

Fixed in commit 0aea9b8562d46ac2a09f2711d8a5a14d2c8a01c6.

W-09	Lack of `EIP-165` interface support validation
Severity	**WARNING**
Status	• ACKNOWLEDGED

Description

The project code interacts with various system components through interfaces, but there is no check to ensure that a contract address supports an interface according to `EIP-165`. This can result in an address being set that does not support the required interface during configuration updates.

Recommendation

We recommend adding checks for interface support when setting contract addresses according to `EIP-165`.

Update

Altitude's response

We will consider this for future releases.

W-10	Potential for duplicate token creation in `TokensFactory`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
TokensFactory.sol	contract `TokensFactory`	21

Description

In the contract `TokensFactory`, any user can create duplicate tokens with nearly identical parameters. This can lead to phishing or spam attacks.

Recommendation

We recommend adding a separate role for addresses that can create token pairs for the `Vault`.

Update

Fixed in commit f7f2fb7195d29fe41db4a11c1ce6a88b7a02731b.

W-11	Deposit limit check may cause transaction reversion in `Ingress`
Severity	**WARNING**
Status	• ACKNOWLEDGED

Location

File	Location	Line
Ingress.sol	contract `Ingress` > function `validateDeposit`	192

Description

In the function `validateDeposit` of contract `Ingress`, a check ensures that after a deposit, the `amount` does not cause the `userBalance` to exceed the `userMaxDepositLimit`. However, this can create potential issues if the `userBalance` after depositing the `amount` exceeds the `userMaxDepositLimit` by a few wei. In this case, the user's transaction will be reverted.

Recommendation

We recommend adding logic to refund the excess amount (`userBalance - userMaxDepositLimit`) to the sender if the `userBalance` exceeds the `userMaxDepositLimit`.

Update

Altitude's response

This logic would be non-trivial to add to the smart contracts but we will mitigate the effects of this in the front-end.

W-12	`increaseAllowance` and `decreaseAllowance` not disabled in `DebtToken`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
DebtToken.sol	contract `DebtToken`	141

Description

In the contract `DebtToken`, the `increaseAllowance` and `decreaseAllowance` functions are not disabled. Since the project uses OpenZeppelin 4.*, the ERC20 implementation also includes these functions.

Recommendation

We recommend disabling the `increaseAllowance` and `decreaseAllowance` functions similarly to how `approve` is disabled.

Update

Fixed in commit d6ba762b6b0c3219bea10b26b3a2d9d412f10de9.

W-13	Lack of support for deflationary tokens in `VaultCore`
Severity	**WARNING**
Status	• ACKNOWLEDGED

Location

File	Location	Line
VaultCore.sol	contract `VaultCore` > function `deposit`	215
VaultERC20.sol	contract `VaultERC20` > function `_postWithdraw`	32

Description

In the mentioned locations, there is no support for deflationary tokens or tokens that may have fees (e.g., `USDC` and `USDT`, which have the possibility to set fees in their code). This can lead to system imbalances.

Recommendation

We recommend adding balance calculations based on the actual amount of tokens received by the contract.

Update

Altitude's response

We will consider this for future releases

W-14	Reassigned `amountTotal` value may bypass zero check in `HarvestableManager`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
HarvestableManager.sol	contract `HarvestableManager` > function `claimRewards`	342

Description

In the function `claimRewards` of contract `HarvestableManager`, there is a check that `amountTotal` is not zero at line #L342:

```
if (amountTotal == 0) {
    revert HM_V1_CLAIM_REWARDS_ZERO();
}
```

However, the value of `amountTotal` is then reassigned at line #L368, and if the values of `amountRequested` and `debtBalance` are both zero, `amountTotal` will also be zero, and the previous condition will be ignored. This can lead to a potential DDOS attack.

Recommendation

Final fix in commit 4d3062752e77322ab555695e57039a7fd9da5d51.

We recommend adding a check that `amountTotal` is not zero after the values have been reassigned.

Update

Altitude's response

How would you define what are 'acceptable' upper and lower bounds `amountTotal` is re-assigned to save gas costs. We care only if the user has rewards or not. If the user has rewards, but `amountRequested` and `debtBalance` are `0`, then nothing will happen.

Oxorio's response

If the user has a reward, but `amountRequested` and `debtBalance` are `0`, the function issues an event. Thus, for example, the user can spam the frontend with messages about zero rewards, spending only gas to complete the transaction.

Altitude's response

Fixed in 4d3062752e77322ab555695e57039a7fd9da5d51.

Oxorio's response

We confirm that commit 4d3062752e77322ab555695e57039a7fd9da5d51 is a valid fix.

W-15	No parameters validation
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
FarmModeDecisionMaker.sol	contract `FarmModeDecisionMaker` > `constructor`	54
FarmModeDecisionMaker.sol	contract `FarmModeDecisionMaker` > `constructor`	55
FarmModeDecisionMaker.sol	contract `FarmModeDecisionMaker` > `constructor`	56
FarmModeDecisionMaker.sol	contract `FarmModeDecisionMaker` > `constructor`	57
FarmModeDecisionMaker.sol	contract `FarmModeDecisionMaker` > function `setFarmDropThreshold`	103
FarmModeDecisionMaker.sol	contract `FarmModeDecisionMaker` > function `setActiveManagement`	117
FarmModeDecisionMaker.sol	contract `FarmModeDecisionMaker` > function `addPriceSources`	133
VaultRegistry.sol	contract `VaultRegistryV1` > function `setProxyAdmin`	337
VaultRegistry.sol	contract `VaultRegistryV1` > function `setInitImpl`	348
VaultOperable.sol	contract `VaultOperable` > `constructor`	28-29
FarmBuffer.sol	contract `FarmBuffer` > `constructor`	21
FarmModeVault.sol	contract `FarmModeVaultV1` > function `setFarmModeConfig`	33
VaultRegistry.sol	contract `VaultRegistryV1` > function `withdrawVaultReserve`	440

Description

In the mentioned locations and other places in the codebase, input parameters are not validated.

Recommendation

We recommend validating input parameters to ensure the provided values are within the required bounds.

Update

Initial fix in commit 11f8c01e0cd001146760baf992be8462fd0d6f13, final fix in commit 5004c48c65aed86c5eddcf10423e20f7160f25fc.

Altitude's response

Fixed in commit 11f8c01e0cd001146760baf992be8462fd0d6f13.

Oxorio's response

After the fix, there are still several areas in the code that require attention.

In the function `setFarmDropThreshold` of contract `FarmModeDecisionMaker`, there is no need to check `_maxThreshold == 0` since this condition has already been verified above.

In the function `hasPriceDiscrepancy` of contract `FarmModeDecisionMaker`, the check is redundant as it is not possible to set an empty `priceSources` array in the contract.

In the function `removePriceSource` of contract `FarmModeDecisionMaker`, when removing an element from `priceSource`, the array cannot be left empty, so it should be considered that `priceSources.length` cannot be equal to `1`.

In the function `initialize` and function `setProxyAdmin` of contract `VaultRegistryV1`, there are still areas without address validation.

Altitude's response

Fixed in 5004c48c65aed86c5eddcf10423e20f7160f25fc.

Oxorio's response

We confirm that commit 5004c48c65aed86c5eddcf10423e20f7160f25fc is a valid fix.

I-01	Redundant `_onlyVault` function in `InterestToken`
Severity	**INFO**
Status	• ACKNOWLEDGED

Location

File	Location	Line
InterestToken.sol	contract `InterestToken` > function `_onlyVault`	43

Description

The `_onlyVault` function in contract `InterestToken` is not used anywhere except for the `onlyVault` modifier.

Recommendation

We recommend removing the redundant function and moving the logic to the `onlyVault` modifier.

Update

Altitude's response

By using a function rather than modifier the overall contract size is reduced (modifier code is replicated everywhere in compiled code, while function logic isn't). We're using the same pattern here for consistency.

I-02	Unused constant `MATH_UNITS` in `InterestToken`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
InterestToken.sol	contract `InterestToken`	30

Description

The constant `MATH_UNITS` in the `InterestToken` contract is not used anywhere else except as a default value for the `interestIndex` variable.

Recommendation

We recommend removing the redundant constant to keep the codebase clean.

Update

Fixed in commit f440d61a298d0181147aca2ce8d842af04278946.

I-03	Fee is charged on withdrawal in `VaultCoreV1`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
VaultCore.sol	contract `VaultCoreV1` > function `_withdraw`	305

Description

In the function `_withdraw` of the contract `VaultCoreV1`, the withdrawal fee is calculated and subtracted from the withdrawal amount. But in the whitepaper the following is stated:

> Altitude does not charge any Origination, Management or Withdraw fees.

Recommendation

We recommend clarifying the fee workflow in the protocol whitepaper or the codebase.

Update

Altitude's response

Withdraw fees charged are temporary and cover deposit fees and avoid certain attack vectors. The documentation has been updated.

I-04	Variable can be immutable in `FarmBufferStrategy`
Severity	**INFO**
Status	• ACKNOWLEDGED

Location

File	Location	Line
FarmBufferStrategy.sol	contract `FarmBufferStrategy`	17

Description

In the `FarmBufferStrategy` contract, the `farmBuffer` variable is set only once in the constructor and cannot be changed thereafter.

Recommendation

We recommend making the `farmBuffer` variable `immutable`.

Update

Altitude's response

This will be fixed in a future version.

I-05	Suboptimal reading of the `harvestStorage.harvests.length` variable from storage in `HarvestableManager`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
HarvestableManager.sol	contract `HarvestableManager` > function `harvest`	25

Description

In the function `harvest` of contract `HarvestableManager`, there are multiple occurrences where the same variable `harvestStorage.harvests.length` is read from storage.

Additionally, after adding data about a new harvest to the `harvestStorage.harvests` array, subtraction of `harvestStorage.harvests.length-1` is used to obtain the previous value:

```
harvestStorage.harvests.push(newHarvest);
snapshots.push(
    CommonTypes.SnapshotType(
        harvestStorage.harvests.length - 1,
        // ...
```

Recommendation

We recommend creating an in-memory variable with the value `harvestStorage.harvests.length` to optimize gas consumption by reading from memory instead of storage and improve code readability.

Update

Fixed in commit 89a52041961424b34652ce243e20109b2c3e9c88.

I-06	Simplifying subtraction of `commit.userHarvestUncommittedEarnings` in `HarvestableManager`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
HarvestableManager.sol	contract `HarvestableManager` > function `_repayLoan`	250

Description

In the function `_repayLoan` of contract `HarvestableManager`, `commit.position.borrowBalance` is subtracted from the variable `realUncommittedEarnings`, but at the same time, the variable `claimableEarnings` is also subtracted, which equals `commit.userHarvestUncommittedEarnings - commit.position.borrowBalance`:

```
uint256 claimableEarnings = commit.userHarvestUncommittedEarnings -
    commit.position.borrowBalance;
realUncommittedEarnings -=
    commit.position.borrowBalance +
    claimableEarnings;
```

In other words, only `commit.userHarvestUncommittedEarnings` could be subtracted from `realUncommittedEarnings`.

Recommendation

We recommend simplifying the subtraction from `realUncommittedEarnings` for codebase cleanliness and gas optimization:

```
realUncommittedEarnings -= commit.userHarvestUncommittedEarnings;
```

Update

Fixed in commit 0535f03984fceeb7ae035bdef255ab527a7e6e7a.

I-07	Code duplication in `HarvestableVaultV1`, `LiquidatableManager`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
LiquidatableManager.sol	contract `LiquidatableManager` > function `_updateEarningsRatio`	150
HarvestableVault.sol	contract `HarvestableVaultV1` > function `_updateEarningsRatio`	199

Description

At the mentioned locations, code duplication of the same function `_updateEarningsRatio` occurs.

Additionally, in the comments of the function in the `HarvestableVaultV1` contract, there is an incorrect reference to `HarvestableManager`, where this duplicate does not exist:

```
/// @dev internal function, duplicated in HarvestableManager
```

Recommendation

We recommend extracting this function into a separate module to avoid code duplication and maintain codebase cleanliness.

Update

Fixed in commit 0535f03984fceeb7ae035bdef255ab527a7e6e7a.

I-08	Use `++i` to save gas
Severity	**INFO**
Status	• FIXED

Description

In all contracts across the codebase `i++` is used in loops. However `++i` costs less gas compared to `i++` or `i += 1` for unsigned integer, as pre-increment is cheaper (about 5 gas per iteration). This statement is true even with the optimizer enabled.

Recommendation

We recommend using pre-increment `++i` instead of post-increment `i++`.

Update

Fixed in commit [9379c5abf23df8c4dc14bc245eb9a979a9c2d512](9379c5abf23df8c4dc14bc245eb9a979a9c2d512).

I-09	Int type initialization to zero is redundant
Severity	**INFO**
Status	• FIXED

Description

In all contracts across the codebase initialization of integer (`int/uint`) type variables to zero is unnecessary. In Solidity, integer variables are automatically initialized to zero by default.

Recommendation

We recommend omitting the explicit initialization of integer variables to zero to streamline the code.

Update

Initial fix in commit 6f431ee2d73c38e0ce3d231b3c58c0a9a412156d, final fix in commit 9624f4030159c4d4e9baa895139d05b66b847962.

Altitude's response

Fixed in commit 6f431ee2d73c38e0ce3d231b3c58c0a9a412156d.

Oxorio's response

One case left.

Altitude's response

Fixed in commit 9624f4030159c4d4e9baa895139d05b66b847962.

Oxorio's response

We confirm that commit 9624f4030159c4d4e9baa895139d05b66b847962 is a valid fix.

I-10	Floating `pragma`
Severity	**INFO**
Status	• FIXED

Description

All contracts across the codebase use the following pragma statement:

```
pragma solidity ^0.8.0;
```

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, either an outdated compiler version that might introduce bugs that affect the contract system negatively or a pragma version too new which has not been extensively tested.

Recommendation

We recommend locking the `pragma` to a specific version of the compiler.

Update

Initial fix in commit d33703393a4bebef55bdc9a873a8b69526b5a440, final fix in commit c9ce21033426d2c5ef716d462a6436e1a8bfc779.

Altitude's response

Fixed in commit d33703393a4bebef55bdc9a873a8b69526b5a440

Oxorio's response

There is one case left in the newly created file.

Altitude's response

Fixed in commit c9ce21033426d2c5ef716d462a6436e1a8bfc779.

Oxorio's response

We confirm that commit c9ce21033426d2c5ef716d462a6436e1a8bfc779 is a valid fix.

I-11	Use `+=` in `CommitMath`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
CommitMath.sol	contract `CommitMath` > function `_calculateHarvestCommit`	197

Description

In the function `_calculateHarvestCommit` of contract `CommitMath` `vaultReserveUncommitted` storage variable is incremented:

```
commit.vaultReserveUncommitted =
    uint256(userHarvestChange - userHarvestChangeNew) +
    commit.vaultReserveUncommitted;
```

Recommendation

We recommend using `+=` statement.

Update

Fixed in commit 1802dba24ea543385c6f1b3464d907eb0e1aec18.

I-12	Manual price limit in `HarvestableManager`
Severity	**INFO**
Status	• NO ISSUE

Location

File	Location	Line
HarvestableManager.sol	contract `HarvestableManager` > function `harvest`	44

Description

In the `harvest` function of the `HarvestableManager` contract, the `price` function parameter is used to limit new harvest price deviation:

```
function harvest(uint256 price) external virtual override {
  ...
  // Check the currently price isn't lower than required
  // @dev this is to guard against sudden price drops
  if (newHarvest.price < price) {
    revert HM_V1_PRICE_TOO_LOW();
  }
```

However, this parameter is set externally and can be different from the actual current price of the asset, which leads to undesirable error.

Recommendation

We recommend using an on-chain oracle to obtain the actual current price of the asset.

Update

Altitude's response

This parameter allows us to guard around sudden price drops (through external manipulation or otherwise) that would significantly affect the health of user positions. This allows us to avoid running a harvest when a significant amount of funds is subject to liquidation. Using an external oracle price would not allow us to achieve this.

I-13	Double execution of `setBalance` logic in `SnapshotableManager`
Severity	**INFO**
Status	• NO ISSUE

Location

File	Location	Line
SnapshotableManager.sol	contract `SnapshotableManager` > function `_commitUser`	127-148

Description

In the `_commitUser` function of the `SnapshotableManager` contract, the `setBalance` logic is executed for a specific `account`:

```
  supplyToken.setBalance(
    ...
  );
  debtToken.setBalance(
    ...
  );

  // If not a partial commit, then update the user's position to account for the latest
interest
  if (snapshotId == snapshots.length) {
    (
      ...
    ) = supplyToken.snapshotUser(account);
    (
      ...
    ) = debtToken.snapshotUser(account);
  }
```

However, the `snapshotUser` function of the `InterestToken` contract (`supplyToken` and `debtToken`) contains the `setBalance` logic as well. Thus, in the case of a full commit, the `setBalance` logic is executed twice.

Recommendation

We recommend moving the `setBalance` calls to an `else` block to avoid the double execution of `setBalance` logic.

Update

Altitude's response

setBalance directly re-assigns balance and index to a user in order to move it further among the commits.

snapshotUser is not using setBalance internally, but takes the last stored balance and index and accumulates interest from that moment up to now.

I-14	Missed error handling in `HarvestableManager`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
HarvestableManager.sol	contract `HarvestableManager` > function `withdrawReserve`	317

Description

In the `withdrawReserve` function of the `HarvestableManager` contract, reserves are withdrawn to the `receiver` address:

```
function withdrawReserve(
  address receiver,
  uint256 amount
) external override returns (uint256) {
  // Calculate amount in reserve
  uint256 maxAmount = IERC20(borrowUnderlying).balanceOf(address(this)) +
    harvestStorage.vaultReserve -
    farmModeStorage.farmModeReserve;

  uint256 readyAmount = maxAmount - harvestStorage.vaultReserve;

  ...

  // Withdraw from the farmStrategy if needed
  if (amount > readyAmount) {
    if (!farmModeStorage.farmMode) {
      // the vaultReserve is part of the vault balance
      // and should be decreased if amount includes it
      uint256 amountDiff = amount - readyAmount;
      harvestStorage.vaultReserve -= amountDiff;
      farmModeStorage.farmModeReserve -= amountDiff;
    } else {
      ...
    }
  }
```

There is an edge case where an unhandled revert occurs. Assume the following values are current:

- `IERC20(borrowUnderlying).balanceOf(address(this))` is `200`
- `vaultReserve` is `100`
- `farmModeReserve` is `50`
- `amount` parameter is `210`
- `farmModeStorage.farmMode` is `false`

Then:

- `maxAmount` is `250`
- `readyAmount` is `150`

An underflow revert occurs in the decreasing of `farmModeReserve`:

```
farmModeStorage.farmModeReserve -= amountDiff;
```

Recommendation

We recommend clearly handling the error when `vaultReserve > farmModeReserve` and `farmMode` is `false`.

Update

Fixed in commit [0bb4b08213aa611958bdf12a18a4d0bab49afe67](0bb4b08213aa611958bdf12a18a4d0bab49afe67).

I-15	Magic numbers
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
CommitMath.sol	contract `CommitMath` > function `_calculateHarvestCommit`	184
CommitMath.sol	contract `CommitMath` > function `_calculateHarvestCommit`	193
CommitMath.sol	contract `CommitMath` > function `_userActiveAssets`	238
CommitMath.sol	contract `CommitMath` > function `_userActiveAssets`	243
HealthFactorCalculator.sol	contract `HealthFactorCalculator` > function `isPositionHealthy`	52
HealthFactorCalculator.sol	contract `HealthFactorCalculator` > function `availableBorrow`	66
HealthFactorCalculator.sol	contract `HealthFactorCalculator` > function `targetBorrow`	92
RebalanceIncentivesController.sol	contract `RebalanceIncentivesController` > function `canRebalance`	98
RebalanceIncentivesController.sol	contract `RebalanceIncentivesController` > function `_validateThresholds`	123
ConfigurableManager.sol	contract `ConfigurableManager` > function `setConfig`	29
ConfigurableManager.sol	contract `ConfigurableManager` > function `setBorrowLimits`	51
ConfigurableManager.sol	contract `ConfigurableManager` > function `setBorrowLimits`	57
ConfigurableManager.sol	contract `ConfigurableManager` > function `setBorrowLimits`	62
GroomableManager.sol	contract `GroomableManager` > function `flashLoanCallback`	104
GroomableVault.sol	contract `GroomableVaultV1` > function `setGroomableConfig`	82
LiquidatableManager.sol	contract `LiquidatableManager` > function `liquidateUsers`	60
LiquidatableManager.sol	contract `LiquidatableManager` > function `liquidateUsers`	69
LiquidatableManager.sol	contract `LiquidatableManager` > function `liquidateUsers`	86
LiquidatableManager.sol	contract `LiquidatableManager` > function `_updateEarningsRatio`	159
LiquidatableVault.sol	contract `LiquidatableVaultV1` > function `setLiquidationConfig`	73

File	Location	Line
LiquidatableVault.sol	contract `LiquidatableVaultV1` > function `setLiquidationConfig`	77
HarvestableManager.sol	contract `HarvestableManager` > function `harvest`	98
HarvestableManager.sol	contract `HarvestableManager` > function `_getVaultActiveAssets`	156
HarvestableVault.sol	contract `HarvestableVaultV1` > function `_updateEarningsRatio`	208
SnapshotableVault.sol	contract `SnapshotableVaultV1` > function `setSnapshotableConfig`	147
VaultCore.sol	contract `VaultCoreV1` > function `calcWithdrawFee`	392
VaultInitializer.sol	contract `VaultCoreV1Initializer` > function `_initializeConfigurableVaultV1`	35
VaultInitializer.sol	contract `VaultCoreV1Initializer` > function `_initializeConfigurableVaultV1`	39
VaultInitializer.sol	contract `VaultCoreV1Initializer` > function `_initializeConfigurableVaultV1`	46
VaultInitializer.sol	contract `VaultCoreV1Initializer` > function `_initializeConfigurableVaultV1`	52
VaultInitializer.sol	contract `VaultCoreV1Initializer` > function `_initializeGroomableVaultV1`	87
VaultInitializer.sol	contract `VaultCoreV1Initializer` > function `_initializeLiquidatableVaultV1`	99
VaultInitializer.sol	contract `VaultCoreV1Initializer` > function `_initializeLiquidatableVaultV1`	103
VaultInitializer.sol	contract `VaultCoreV1Initializer` > function `_initializeSnapshotableVaultV1`	115
FarmModeDecisionMaker.sol	contract `FarmModeDecisionMaker` > function `hasPriceDiscrepancy`	199

Description

In the mentioned locations literal values with unexplained meaning are used to perform calculations.

Recommendation

We recommend defining a constant for every magic number, giving it a clear and self-explanatory name.

Update

Fixed in commit a55589aa477777c15c0da62216a1c510be71afc6.

3 APPENDIX

OXORIO

3.1 DISCLAIMER

At the request of client, Oxorio consents to the public release of this audit report. The information contained in this audit report is provided "as is," without any representations or warranties whatsoever. Oxorio disclaims any responsibility for damages that may arise from or in relation to this audit report. Oxorio retains copyright of this report.

The audit makes no statements or warranties about the utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about the fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

3.2 SECURITY ASSESSMENT METHODOLOGY

Oxorio's smart contract audit methodology is designed to ensure the security, reliability, and compliance of smart contracts throughout their development lifecycle. Our process integrates the Smart Contract Security Verification Standard (SCSVS) with our advanced techniques to address complex security challenges. For a detailed look at our approach, please refer to the full version of our methodology. Here is a concise overview of our auditing process:

1. Project Architecture Review

All necessary information about the smart contract is gathered, including its intended functionality and dependencies. This stage sets the foundation by reviewing documentation, business logic, and initial code analysis.

2. Vulnerability Assessment

This phase involves a deep dive into the smart contract's code to identify security vulnerabilities. Rigorous testing and review processes are applied to ensure robustness against potential attacks.

This stage is focused on identifying specific vulnerabilities within the smart contract code. It involves scanning and testing the code for known security weaknesses and patterns that could potentially be exploited by malicious actors.

3. Security Model Evaluation

The smart contract’s architecture is assessed to ensure it aligns with security best practices and does not introduce potential vulnerabilities. This includes reviewing how the contract integrates with external systems, its compliance with security best practices, and whether the overall design supports a secure operational environment.

This phase involves a analysis of the project's documentation, the consistency of business logic as documented versus implemented in the code, and any assumptions made during the design and development phases. It assesses if the contract's architectural design adequately addresses potential threats and integrates necessary security controls.

4. Cross-Verification by Multiple Auditors

Typically, the project is assessed by multiple auditors to ensure a diverse range of insights and thorough coverage. Findings from individual auditors are cross-checked to verify accuracy and completeness.

5. Report Consolidation

Findings from all auditors are consolidated into a single, comprehensive audit report. This report outlines potential vulnerabilities, areas for improvement, and an overall assessment of the smart contract's security posture.

6. Reaudit of Revised Submissions

Post-review modifications made by the client are reassessed to ensure that all previously identified issues have been adequately addressed. This stage helps validate the effectiveness of the fixes applied.

7. Final Audit Report Publication

The final version of the audit report is delivered to the client and published on Oxorio's official website. This report includes detailed findings, recommendations for improvement, and an executive summary of the smart contract's security status.

3.3 CODEBASE QUALITY ASSESSMENT REFERENCE

The tables below describe the codebase quality assessment categories and rating criteria used in this report.

Category	Description
Access Control	Evaluates the effectiveness of mechanisms controlling access to ensure only authorized entities can execute specific actions, critical for maintaining system integrity and preventing unauthorized use.
Arithmetic	Focuses on the correct implementation of arithmetic operations to prevent vulnerabilities like overflows and underflows, ensuring that mathematical operations are both logically and semantically accurate.
Complexity	Assesses code organization and function clarity to confirm that functions and modules are organized for ease of understanding and maintenance, thereby reducing unnecessary complexity and enhancing readability.
Data Validation	Assesses the robustness of input validation to prevent common vulnerabilities like overflow, invalid addresses, and other malicious input exploits.
Decentralization	Reviews the implementation of decentralized governance structures to mitigate insider threats and ensure effective risk management during contract upgrades.
Documentation	Reviews the comprehensiveness and clarity of code documentation to ensure that it provides adequate guidance for understanding, maintaining, and securely operating the codebase.
External Dependencies	Evaluates the extent to which the codebase depends on external protocols, oracles, or services. It identifies risks posed by these dependencies, such as compromised data integrity, cascading failures, or reliance on centralized entities. The assessment checks if these external integrations have appropriate fallback mechanisms or redundancy to mitigate risks and protect the protocol's functionality.
Error Handling	Reviews the methods used to handle exceptions and errors, ensuring that failures are managed gracefully and securely.
Logging and Monitoring	Evaluates the use of event auditing and logging to ensure effective tracking of critical system interactions and detect potential anomalies.
Low-Level Calls	Reviews the use of low-level constructs like inline assembly, raw `call` or `delegatecall`, ensuring they are justified, carefully implemented, and do not compromise contract security.

Category	Description
Testing and Verification	Reviews the implementation of unit tests and integration tests to verify that codebase has comprehensive test coverage and reliable mechanisms to catch potential issues.

3.3.1 Rating Criteria

Rating	Description
Excellent	The system is flawless and surpasses standard industry best practices.
Good	Only minor issues were detected; overall, the system adheres to established best practices.
Fair	Issues were identified that could potentially compromise system integrity.
Poor	Numerous issues were identified that compromise system integrity.
Absent	A critical component is absent, severely compromising system safety.
Not Applicable	This category does not apply to the current evaluation.

3.4 FINDINGS CLASSIFICATION REFERENCE

3.4.1 Severity Level Reference

The following severity levels were assigned to the issues described in the report:

Title	Description
CRITICAL	Issues that pose immediate and significant risks, potentially leading to asset theft, inaccessible funds, unauthorized transactions, or other substantial financial losses. These vulnerabilities represent serious flaws that could be exploited to compromise or control the entire contract. They require immediate attention and remediation to secure the system and prevent further exploitation.
MAJOR	Issues that could cause a significant failure in the contract's functionality, potentially necessitating manual intervention to modify or replace the contract. These vulnerabilities may result in data corruption, malfunctioning logic, or prolonged downtime, requiring substantial operational changes to restore normal performance. While these issues do not immediately lead to financial losses, they compromise the reliability and security of the contract, demanding prioritized attention and remediation.
WARNING	Issues that might disrupt the contract's intended logic, affecting its correct functioning or making it vulnerable to Denial of Service (DDoS) attacks. These problems may result in the unintended triggering of conditions, edge cases, or interactions that could degrade the user experience or impede specific operations. While they do not pose immediate critical risks, they could impact contract reliability and require attention to prevent future vulnerabilities or disruptions.
INFO	Issues that do not impact the security of the project but are reported to the client's team for improvement. They include recommendations related to code quality, gas optimization, and other minor adjustments that could enhance the project's overall performance and maintainability.

3.4.2 Status Level Reference

Based on the feedback received from the client's team regarding the list of findings discovered by the contractor, the following statuses were assigned to the findings:

Title	Description
NEW	Waiting for the project team's feedback.

Title	Description
FIXED	Recommended fixes have been applied to the project code and the identified issue no longer affects the project's security.
ACKNOWLEDGED	The project team is aware of this finding. Recommended fixes for this finding are planned to be made. This finding does not affect the overall security of the project.
NO ISSUE	Finding does not affect the overall security of the project and does not violate the logic of its work.

3.5 ABOUT OXORIO

OXORIO is a blockchain security firm that specializes in smart contracts, zk-SNARK solutions, and security consulting. With a decade of blockchain development and five years in smart contract auditing, our expert team delivers premier security services for projects at any stage of maturity and development.

Since 2021, we've conducted key security audits for notable DeFi projects like Lido, 1Inch, Rarible, and deBridge, prioritizing excellence and long-term client relationships. Our co-founders, recognized by the Ethereum and Web3 Foundations, lead our continuous research to address new threats in the blockchain industry. Committed to the industry's trust and advancement, we contribute significantly to security standards and practices through our research and education work.

Our contacts:

- oxor.io
- ping@oxor.io
- Github
- Linkedin
- Twitter

THANK YOU FOR CHOOSING