FATHOM VAULTS SMART CONTRACTS

OXORIO

MARCH 28, 2024

CONTENTS

1 INTRO

1.1 DISCLAIMER

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.

1.2 ABOUT OXORIO

Oxorio is a young but rapidly growing audit and consulting company in the field of the blockchain industry, providing consulting and security audits for organizations from all over the world. Oxorio has participated in multiple blockchain projects during which smart contract systems were designed and deployed by the company.

Oxorio is the creator, maintainer, and major contributor of several blockchain projects and employs more than 5 blockchain specialists to analyze and develop smart contracts.

Our contacts:

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

1.3 SMART CONTRACTS AUDIT AND SECURITY ASSESSMENT METHODOLOGY

When conducting a smart contracts audit, the audit team follows a structured approach to systematically review, identify, and address potential vulnerabilities in the codebase. The goal is to ensure the integrity, security, and robustness of the smart contracts. Here is an overview of the audit team's approach:

1. **Scope Definition**
   The audit team initiates the process by clearly defining the scope of the audit. This involves understanding the architecture, functionalities, and dependencies of the smart contracts under review. The scope also includes assessing the integration with external systems, oracles, and other relevant components.

2. **Code Review**
   The heart of the audit process is a meticulous review of the smart contracts' codebase. The team examines each contract for vulnerabilities such as reentrancy issues, arithmetic overflows/underflows, and unexpected logic flows. The goal is to ensure that the code functions as intended and to identify potential areas of exploitation.

3. **Checking Against Security Best Practices**
   The audit team checks for adherence to security best practices in smart contract development. This includes proper input validation, use of safe arithmetic operations, protection against reentrancy attacks, and secure handling of user funds. The team also evaluates whether the code follows established coding standards and guidelines.

4. **External Dependency Analysis**
   Smart contracts often interact with external systems, oracles, and third-party components. The audit team thoroughly analyzes these dependencies to identify potential vulnerabilities or points of failure. Special attention is given to ensuring that external inputs are validated and trusted sources are used.

5. **Gas Optimization**
   Efficient gas usage is critical for smart contracts, especially in decentralized environments where transaction costs matter. The audit team evaluates the gas efficiency of the code, looking for opportunities to optimize computations, reduce storage costs, and improve overall contract performance.

6. **Test Coverage**
   The audit team assesses the comprehensiveness of the test suite associated with the smart contracts. A robust test suite is essential for detecting and preventing

regressions. The team may also conduct additional testing, including edge case scenarios and simulations of potential attack vectors.

7. **Security Tools and Automated Analysis**
   Security tools and automated analysis are employed to complement manual reviews. These tools help identify common vulnerabilities, such as code duplication, insecure dependencies, and potential issues that might be overlooked during manual inspection.

8. **Documentation Review**
   Documentation is an integral part of smart contract development. The audit team reviews documentation to ensure that it accurately reflects the code's functionalities, security considerations, and usage instructions. Clear and comprehensive documentation contributes to the overall transparency of the project.

9. **Result's Cross-Check by Different Auditors**
   Following this initial individual assessment, a crucial step ensues – the mutual cross-check process. During this collaborative stage, the audit results are meticulously compared and verified among the different auditors involved, each contributing their unique expertise and perspectives.

10. **Report Consolidation**
    Once the individual audits are completed, the next step involves consolidating the audited reports from the multiple auditors. This consolidation process involves compiling and integrating the findings, recommendations, and insights from each auditor into a unified and cohesive document. The consolidated report captures the collective expertise and assessments of the audit team and presents a comprehensive overview of the project's security status.

11. **Reaudit of New Editions**
    After the client has received and reviewed the initial audit report, addressing any identified issues and implementing necessary fixes, a crucial step follows – the reaudit of new editions. During this phase, the audit team conducts a thorough reevaluation, double-checking the previously identified issues and verifying the effectiveness of the implemented fixes. The results of this reaudit are then incorporated into a new version of the audit report, providing an updated and accurate representation of the project's security posture.

12. **Final Audit Report Publication**
    The culmination of the audit process involves the publication of the final audit report. In this stage, the conclusive version of the audit report is provided to the client, offering a comprehensive summary of the project's security strengths, vulnerabilities, and recommended improvements. Simultaneously, the finalized audit report is made publicly accessible by being published on the official website of the auditing company. This transparency not only fosters accountability but also serves as a valuable resource for the broader community, allowing stakeholders and the public

to gain insights into the security measures and practices employed by the audited project.

By following this comprehensive approach, the audit team aims to contribute to the development of secure and reliable smart contracts, instilling confidence in the integrity of the decentralized ecosystem.

1.4 FINDINGS CLASSIFICATION

1.4.1 Severity Level Reference

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

- **CRITICAL**: A bug leading to assets theft, locked fund access, or any other loss of funds due to transfer to unauthorized parties.
- **MAJOR**: A bug that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
- **WARNING**: A bug that can break the intended contract logic or expose it to DDoS attacks.
- **INFO**: Minor issue or recommendation reported to / acknowledged by the client's team.

1.4.2 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:

- **NEW**: Waiting for the project team's feedback.
- **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.
- **DISMISSED**: The issue or recommendation was dismissed by the client.

2 AUDITING FATHOM VAULTS

OXORIO

In the fast-paced world of decentralized finance (DeFi), security is paramount. With the rise of innovative yield aggregator protocols like Fathom Vaults, users and developers alike understand the critical importance of rigorous audits to ensure the integrity and safety of the platform. Here we describe the audit process of Fathom Vaults, shedding light on how the audit team worked to discover and mitigate possible vulnerabilities.

2.1 UNDERSTANDING FATHOM VAULTS

Fathom Vaults, like many yield aggregator protocols, aims to optimize returns for users by automatically shifting funds among various liquidity pools and yield farming opportunities in the DeFi ecosystem. The audit of Fathom Vaults was undertaken to identify and address potential vulnerabilities that could compromise the security of users' funds.

Project overview

Fathom Vault is an ERC-4626 compliant contract that takes in user deposits, mints shares corresponding to the user's share of the underlying assets held in that vault, and then allocates the underlying asset to a range of different "strategies" that earn yield on that asset.

A strategy refers to a yield-generating contract added to a vault that has the needed ERC-4626 interface. The strategy takes the underlying asset and deploys it to a single source, generating yield on that asset.

`TokenizedStrategy` is technical implementation of a Strategy that is also a stand-alone ERC4626 compliant Vault. These are the yield generators in the Fathom ecosystem. This pattern can be used so that either Vaults or individual users can deposit directly into and receive shares in return.

Vault Factory is a factory contract that ensures that Vault contracts can be easily and trustlessly deployed from it.

Users mint shares by depositing tokens, expecting passive yield with acknowledged potential loss. Users can redeem shares for underlying tokens. To address price per share (pps) volatility, the Vault deploys strategic mechanisms:

- Internal Accounting: tracks debt and idle state through internal accounting instead of balanceOf().
- Profit Locking Mechanism: Fathom Vaults' mechanism locks profits by issuing burnable shares over an unlock period.
- Loss and Fee Mitigation: the Vault offsets losses or fees by burning owned locked shares.

2.2 HIGH-LEVEL FLOW OF FUNDS

1. **User Deposit**
   Users initiate the process by depositing their funds into the Fathom Vault using `deposit` function. Deposited funds are pooled together with other users' funds. This creates a collective pool of assets that the yield aggregator will use to generate return. Total amount of available funds in the vault is kept in the `totalIdle` variable

2. **Strategy Selection**
   Fathom Vault employs a set of strategies to maximize returns on the pooled funds. Strategies can include yield farming, liquidity provision, lending, and other DeFi mechanisms. The protocol implements generic `TokenizedStrategy` that is itself an ERC4626 compliant single strategy Vault. The funds are sent to the strategy by the user with a `STRATEGY_MANAGER` role with a `updateDebt` call which caluculates amount of funds available for deposit and calls the `deposit` method of the strategy.

3. **Strategy Execution and Accruing Returns**
   The chosen strategies are executed on different DeFi protocols. This may involve providing liquidity to decentralized exchanges, lending assets on lending platforms, or participating in yield farming on various protocols. As the strategies generate returns, profits accrue to the pooled funds. Returns may come in the form of interest, trading fees, or additional tokens earned through yield farming. To account for the profit the method `harvestAndReport` of the strategy is called, which updates the amount of funds available on the strategy, calculate appropriate fees and proceedes with the mechanism of profit locking.

4. **User Redemption or Withdrawal**
   Users have the option to redeem or withdraw their funds along with any accrued returns. This process involves converting their share of the pooled funds back into the original asset. This is achieved by the call to `withdraw` method which will use idle funds of the vault if sufficient, or will withdraw funds from the strategies together with accounting for the loss if present.

2.3 RESEARCHED ATTACK VECTORS

Fathom Vaults as a yield aggregator protocol is susceptible to various attack vectors due to the complex smart contract interactions and the constantly evolving nature of the DeFi space. Here are some common attack vectors on yield aggregator protocols:

1. **Flash Loan Attacks**
   Attackers exploit the ability to borrow a large sum of funds temporarily using flash loans and manipulate the protocol within a single transaction. This can be used to exploit vulnerabilities, such as front-running or price manipulation.

2. **Reentrancy Attacks**
   Smart contracts are susceptible to reentrancy 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.

3. **Liquidity Pool Exploitation**
   Attackers may take advantage of vulnerabilities in the underlying liquidity pools. This could involve manipulating token prices, exploiting weaknesses in automated market makers (AMMs), or exploiting vulnerabilities in the way liquidity is managed.

4. **Smart Contract Bugs**
   Coding errors or vulnerabilities in the smart contracts themselves can be exploited by attackers. This includes issues like arithmetic overflows/underflows, unexpected logic flows, or insecure code practices that can lead to unintended consequences.

5. **Front-Running**
   Attackers can front-run transactions by anticipating and executing trades before a legitimate transaction is confirmed, exploiting price changes to their advantage. This is particularly relevant in DeFi protocols where transactions are publicly visible before being mined.

6. **Supply Chain Attacks**
   Malicious actors might compromise the external dependencies of a yield aggregator, such as external contracts, libraries, or tools. This can introduce vulnerabilities or malicious code into the protocol.

7. **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.

2.4 CODE ANALYSIS PROCESS

During the analysis, sections of code that were of particular interest for the audit were identified.

Since the `Fathom Vault` project is based on `Yearn V3`, special attention was paid to code sections that differ between `yearn` and `Fathom Vaults`. This code potentially could contain copying errors and may not be covered by tests.

It was also important to consider that the original `Yearn` is written in `Vyper`, while `Fathom Vault` is written in `Solidity`. Therefore, the code taken from `Yearn` had to be checked for issues specific to these languages.

The most critical parts of the code in terms of security, volume, and logic complexity are related to fund withdrawal (`withdraw`, `redeem`) and report calculation (`processReport`).

Additionally, the mechanism for recalculating unlocked shares could present a significant potential for errors due to its complexity.

2.4.1 Strategy Report Generation

During the report generation process, it is crucial to correctly calculate gain and loss. It is unacceptable for the report processing to result in the issuance or burning of more shares than necessary, considering the fees. Potential weak points here could include:

- Mathematical calculations and their sequence, which may lead to an incorrect ratio of assets and shares in the vault.
- Rounding during division, which introduces inaccuracies in calculations and may result in exceeding limits, leading to report generation errors.

Particular attention should be paid to the differences in share unlocking processes between scenarios of significant profit and extensive share burning compared to scenarios of small profit and minimal share burning. This is to address the potential manipulation of share unlocking rates (`profitUnlockingRate`) or exceeding the available share size for unlocking.

```
// newProfitLockingPeriod is a weighted average between the remaining time of the previously
locked shares and the profitMaxUnlockTime
uint256 newProfitLockingPeriod = (previouslyLockedTime + newlyLockedShares *
profitMaxUnlockTime) / totalLockedShares;
// Calculate how many shares unlock per second.
profitUnlockingRate = (totalLockedShares * MAX_BPS_EXTENDED) / newProfitLockingPeriod;
```

2.4.2 Fund Withdrawal

A special role in the audit process was played by the analysis of fund withdrawal functions. The main risk associated with them is the possibility of withdrawing more funds than are available to the user. Therefore, the audit focused on the potential exceeding of the withdrawal limit:

```
uint256 maxWithdrawAmount = _maxWithdraw(owner, maxLoss, _strategies);
if (assets > maxWithdrawAmount) {
    revert ExceedWithdrawLimit(maxWithdrawAmount);
}
```

Here, attention should be paid to inaccuracies in division rounding and potential mathematical errors. For example, the code involves many conversions from assets to shares and vice versa, using both rounding up and rounding down:

```
uint256 shares = _convertToShares(assets, Rounding.ROUND_UP);
// ...
uint256 maxAssets = _convertToAssets(sharesBalanceOf[owner], Rounding.ROUND_DOWN);
```

Due to the interaction with the `withdrawLimitModule` and the strategy contract to obtain limits, it was also necessary to analyze possible read-only reentrancy attacks.

Another potential attack vector is inflation attacks, so it was necessary to check the possibility of manipulating contract balances to create an incorrect ratio of shares to assets in the vault.

3 AUDIT SCOPE

The scope of this audit includes smart contracts at the main folder.

The audited commit identifier is `43712da89b18c70ca13ad6fd7d7b5bc70fbf11db`.

4. FINDINGS REPORT

OXORIO

4.1 CRITICAL

C-01	Conversion of losses and fees into shares occurs after changing `totalDebt` in `VaultPackage`
Severity	CRITICAL
Status	• FIXED

Location

File	Location	Line
VaultPackage.sol	contract `VaultPackage` > function `_calculateShareManagement`	1449

Description

In the function `_calculateShareManagement` of the contract `VaultPackage`, the process of modifying the `totalAssets` occurs before calculating `shareToBurn`. This results in the conversion of loss into `shareToBurn` being computed based on the `PPS` (price per share), already accounting for the incurred loss.

To simplify, let's consider a scenario without considering fees. Assume `totalAssets=100` tokens, `totalSupply=100` shares, of which `50` are locked shares held by the contract. Now, let's say we incurred a loss of `30` tokens. To maintain `PPS=1`, we need to burn `30` shares.

In the `_calculateShareManagement` function, we first reduce `totalDebt` by `30` tokens. Now, `totalAssets = 100-30 = 70`.

Next, we call the `_convertToShares` function to convert the `30` token loss into shares. Inside the `_convertToShares` function, with `assets=30`, `currentTotalSupply=100`, and `currentTotalAssets=70`, the function returns `30 * 100 / 70 + 1 = 43` shares to burn:

```
uint256 numerator = assets * currentTotalSupply;
uint256 shares = numerator / currentTotalAssets;
if (rounding == Rounding.ROUND_UP && numerator % currentTotalAssets != 0) {
    shares += 1;
}
```

Thus, instead of burning `30` shares, we burn `43`, causing the new `PPS` to be `70/(100-43) = 1.228` instead of `1`.

The calculation of `shares.accountantFeesShares` and `shares.protocolFeesShares` in case of protocol gain is affected in similar way - the accountant and the protocol will get less fees than they should.

Recommendation

We recommend calculating the number of shares for an asset using the `_convertToShares` function before modifying `totalAssets`.

Update

Client's response

Fixed in commit `dbb492893822e87c36c5ccdec83951739c8d3930`.

M-01	Incompatibility with deflationary tokens
Severity	MAJOR
Status	• ACKNOWLEDGED

Location

File	Location	Line
TokenizedStrategy.sol	contract `TokenizedStrategy` > function `_deposit`	1304
VaultPackage.sol	contract `VaultPackage` > function `_deposit`	1098

Description

In the function `_deposit` of the contract `TokenizedStrategy` and the function `_deposit` of the contract `VaultPackage` the `_erc20SafeTransferFrom` fuction call may result in fewer tokens transferred to the contract due to fees taken during call by certain tokens. Tokens with fee-on-transfer functionality, like USDT, which is not currently activated, may cause incorrect accounting in the protocol if used as assets. Using rebase tokens as protocol assets may lead to a mismatch between the vault or strategy balance and storage balance values.

Recommendation

We recommend using balance differences for fee accounting in the case of fee-on-transfer tokens, account for the changes in balance due to operations in the strategy and account for changes in balances of deflationary tokens if utilized in the protocol.

Update

Client's response

We introduced the new mechanism of clearly stating asset type (uint256 public assetType; // 1 for Normal / 2 for Deflationary / 3 for Rebasing). Before performing the transfer, we check for the asset type and make a transfer based on that. For this version of the vault, we support only the first type - Normal asset.

4.3 WARNING

W-01	Parameter validation in `VaultPackage`
Severity	WARNING
Status	• FIXED

Location

File	Location	Line
VaultPackage.sol	contract `VaultPackage` > function `initialize`	50

Description

In the function `initialize` of the contract `VaultPackage` the parameter `_asset` is not validated to be a non-zero address.

Recommendation

We recommend adding validation for the function parameter.

Update

Client's response

The issue is fixed in commit `b194ee12319f52be963ac5a8cd5342b7ef69472a`.

W-02	No validation for duplicate strategies in `VaultPackage`
Severity	WARNING
Status	• FIXED

Location

File	Location	Line
VaultPackage.sol	contract `VaultPackage` > function `setDefaultQueue`	83

Description

In the function `setDefaultQueue` of the contract `VaultPackage` no validation is performed for duplicate strategies in the new queue passed in the `newDefaultQueue` parameter. Attempting to withdraw funds twice from the same strategy may result in accounting errors.

Recommendation

We recommend validating the withdraw queue for duplicate strategies.

Update

Client's response

The issue is fixed in commit `b194ee12319f52be963ac5a8cd5342b7ef69472a`.

W-03	Wrong rounding of `lossesUserShare` in `VaultPackage`
Severity	WARNING
Status	• FIXED

Location

File	Location	Line
VaultPackage.sol	contract `VaultPackage` > function `_assessShareOfUnrealisedLosses`	1708

Description

In the function `_assessShareOfUnrealisedLosses` of the contract `VaultPackage`, the calculation of `lossesUserShare` involves integer division of `numerator` by `strategyCurrentDebt`. This leads to `lossesUserShare` being either exact or slightly larger than expected. In cases of division with a remainder, an additional 1 token is added to `lossesUserShare`:

```
uint256 numerator = assetsNeeded * strategyAssets;
uint256 lossesUserShare = assetsNeeded - numerator / strategyCurrentDebt;

// Always round up.
if (numerator % strategyCurrentDebt != 0) {
    lossesUserShare += 1;
}
```

This results in withdrawing `1` token unit less than intended from the strategy in the `_withdrawAssets` function using `_withdrawFromStrategy`. In scenarios where we intend to withdraw only `1` token, an underflow error may occur.

For instance, if `strategyCurrentDebt=100` and `strategyAssets=90`, and intending to withdraw only `1` token (`assetsNeeded=1`), `lossesUserShare = 1 - (1*90) / 100 = 1`. Consequently, the function returns `2` tokens because `numerator % strategyCurrentDebt = 90 % 100 !=0`.

In the subsequent `_withdrawAssets` function, an underflow error will occur due to `assetsToWithdraw=1` and `unrealisedLossesShare=2`:

```
uint256 unrealisedLossesShare = _assessShareOfUnrealisedLosses(strategy, assetsToWithdraw);
if (unrealisedLossesShare > 0) {
    if (currMaxWithdraw < assetsToWithdraw - unrealisedLossesShare) {
```

Recommendation

We recommend removing additional rounding from the function.

Update

Client's response

That was there originally because Yearn had it here: yearn/yearn-vaults-v3@d8abf37/contracts/VaultV3.vy#L731

This rounding is part of a larger function that calculates the share of unrealized losses a user would bear if they were to withdraw assets from a strategy that has experienced a loss, ensuring that losses are distributed fairly among all participants based on their share of the investment.

After reviewing it one more time, I think we actually need to keep that because If the calculation of a user's share of the loss resulted in a fractional value, simply truncating this number (rounding down) could lead to a situation where the total accounted loss across all users is less than the actual loss. This discrepancy would unfairly distribute the strategy's total loss, leaving a portion of it unaccounted for.

Rounding up ensures that every tiny fraction of a loss is accounted for and attributed to the users. This might seem disadvantageous from a user's perspective, as it could slightly increase their share of the loss, but it's a fairer approach when considering the collective responsibility for the losses incurred by the strategy.

Oxorio's response

You write the following: "If the calculation of a user's share of the loss resulted in a fractional value, simply truncating this number (rounding down) could lead to a situation where the total accounted loss across all users is less than the actual loss."

But rounding down during the division `numerator / strategyCurrentDebt` leads to an increase in losses instead of reducing them because subtraction occurs:

```
uint256 lossesUserShare = assetsNeeded - numerator / strategyCurrentDebt
```

Let's consider an example. Suppose `3` users contribute `100` asset tokens each to the protocol. The current debt of the strategy is `strategyCurrentDebt = 300`, and the strategy incurs losses of `100` asset tokens, leaving `strategyAssets = 200`.

Then each of the users incurs a loss of `lossesUserShare = 34`:

```
// numerator = 100 * 200 = 20000
uint256 numerator = assetsNeeded * strategyAssets;
// lossesUserShare = 100 - 20000 / 300 = 34
uint256 lossesUserShare = assetsNeeded - numerator / strategyCurrentDebt;
```

At this point, the total losses for all users amount to `34 * 3 = 102`, which is more than the actual losses incurred by the strategy:

```
strategyCurrentDebt - strategyAssets = 300 - 200 = 100
```

However, the code adds an additional `+1` to the loss if the numerator is divided by the debt with a remainder:

```
if (numerator % strategyCurrentDebt != 0) {
    lossesUserShare += 1;
}
```

This means each user will incur a loss of `35`, resulting in the total loss for users being `35 * 3 = 105`.

Client's response

The issue is fixed in commit `94e05c538d51acd16008680de038791e10d18b3d`.

W-04	`maxRedeem` returns more shares than redeemable in `VaultPackage`
Severity	WARNING
Status	• FIXED

Location

File	Location	Line
VaultPackage.sol	contract `VaultPackage` > function `maxRedeem`	648

Description

In the function `maxRedeem` of the contract `VaultPackage`, the function `_convertToShares` is invoked with the parameter `Rounding.ROUND_UP`:

```
function maxRedeem(address owner, uint256 maxLoss, address[] calldata _strategies) external
view override returns (uint256) {
    uint256 maxWithdrawAmount = _maxWithdraw(owner, maxLoss, _strategies);
    uint256 sharesEquivalent = _convertToShares(maxWithdrawAmount, Rounding.ROUND_UP);
    return Math.min(sharesEquivalent, sharesBalanceOf[owner]);
}
```

This leads to the `maxRedeem` function returning an inflated value of the maximum shares redeemable, resulting in a revert when attempting to redeem this number of shares.

For example, if `maxWithdrawAmount=1999` tokens are passed to the `_convertToShares` function with `totalAssets=3000` tokens and `totalSupply=1999` shares, the `_convertToShares` function with `Rounding.ROUND_UP` returns `1999 * 1999 / 3000 +1 = 1333` shares.

We obtained the maximum number of shares for redemption, which is `1333` shares. Now, we call the `redeem` function with this number of shares.

First, the shares are converted to assets:

```
uint256 assets = _convertToAssets(shares, Rounding.ROUND_DOWN);
```

As a result of this conversion, we get `assets = 1333 * 3000 / 1999 = 2000` tokens.

Next, inside the `_redeem` function, a revert occurs because `assets=2000` and `maxWithdrawAmount=1999`, similar to when calling the `maxRedeem` function. In other words, `assets > maxWithdrawAmount`:

```
uint256 maxWithdrawAmount = _maxWithdraw(owner, maxLoss, _strategies);
if (assets > maxWithdrawAmount) {
    revert ExceedWithdrawLimit(maxWithdrawAmount);
}
```

Recommendation

We recommend converting tokens to shares with rounding down in the `maxRedeem` function to avoid discrepancies leading to reverts during redemption.

Update

Client's response

The issue is fixed in commit `b194ee12319f52be963ac5a8cd5342b7ef69472a`.

W-05	Using `msg.sender` instead of the dedicated `sender` parameter in `VaultPackage`
Severity	WARNING
Status	• FIXED

Location

File	Location	Line
VaultPackage.sol	contract `VaultPackage` > function `_deposit`	1096

Description

In the function `_deposit` of contract `VaultPackage`, `msg.sender` is used as the source address for the transfer, while the `sender` parameter is only used when emitting an event. However, the function itself is called only once within the `deposit` function, where the `sender` parameter is set to `msg.sender`.

Recommendation

We recommend using the `sender` parameter instead of `msg.sender` within the `_deposit` function.

Update

Client's response

The issue is fixed in commit `b194ee12319f52be963ac5a8cd5342b7ef69472a`.

4.4 INFO

I-01	Redundant code in `VaultPackage`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
VaultPackage.sol	contract `VaultPackage` > function `_revokeStrategy`	1396

Description

In the function `_revokeStrategy` of the contract `VaultPackage` the strategy is removed from the default queue in a two step procedure: verifying whether the strategy is present in the queue and copying the old queue to the new queue, skipping the revoked strategy. Assuming that all strategies that receive the funds from the vault should be in the queue for funds be withdrawable, the step of verifiying the presence of the strategy the queue is redundant.

Recommendation

We recommend removing redundant verification to reduce complexity, maintain code cleanliness, and decrease the gas consumption.

Update

Client's response

Fixed in commit `8e0973eed49f2dd41e03012161f17280d9965d21`.
The function was optimized.

I-02	Method can be front-runned to avoid loss in `VaultPackage`
Severity	INFO
Status	• ACKNOWLEDGED

Location

File	Location	Line
VaultPackage.sol	contract `VaultPackage` > function `processReport`	273

Description

In the function `processReport` of the contract `VaultPackage` the amount of loss for the strategy is calculated and immediately factored into the price per share. In the event the strategy incurs a loss, a malicious protocol user may choose to front-run the `processReport` function to withdraw funds and avoid negative impact on their shares. The funds can be deposited back into the protocol right after the `processReport` transaction.

Recommendation

We recommend considering the possibility of front-running and take measures if critical for the protocol. The `processReport` transaction can be posted with private pool, and withdrawals from the protocol may be organized in such way that makes withdrawals prior to loss socialization impossible.

Update

Client's response

The issue will be fixed in future releases.

I-03	Redundant code in `VaultPackage`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
VaultPackage.sol	contract `VaultPackage` > function `initialize`	42

Description

In the function `initialize` of contract `VaultPackage`, line 42 is a duplicate of line 41.

Recommendation

We recommend removing redundant code.

Update

Client's response

The issue is fixed in commit `b194ee12319f52be963ac5a8cd5342b7ef69472a`.

I-04	Redundant check for `decimalsValue` in `VaultPackage`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
VaultPackage.sol	contract `VaultPackage` > function `initialize`	52

Description

In the function `initialize` of the contract `VaultPackage` there is a redundant validation `decimalsValue < 256`, which is always true for `uint8` type.

Recommendation

We recommend removing redundant validation.

Update

Client's response

The issue is fixed in commit `089c7c823f5d2763034db378470901c16283ebaf`.

I-05	Use addition and subtraction assignement operators to improve code readability in `VaultPackage`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
VaultPackage.sol	contract `VaultPackage` > function `_transfer`	980
VaultPackage.sol	contract `VaultPackage` > function `_transfer`	982

Description

In the function `_transfer` of the contract `VaultPackage` the code readability can be improved by using addition and subtraction assignement operators.

Recommendation

We recommend using addition and subtraction assignement operators to improve code readability and save gas.

Update

Client's response

The issue is fixed in commit `089c7c823f5d2763034db378470901c16283ebaf`.

I-06	Naming can be improved in `VaultStorage`
Severity	INFO
Status	• NO ISSUE

Location

File	Location	Line
VaultStorage.sol	contract `VaultStorage`	32
VaultStorage.sol	contract `VaultStorage`	63
VaultStorage.sol	contract `VaultStorage`	75
VaultStorage.sol	contract `VaultStorage`	78
VaultStorage.sol	contract `VaultStorage`	80
VaultStorage.sol	contract `VaultStorage`	89
VaultStorage.sol	contract `VaultStorage`	91

Description

In the mentioned locations the naming of variables can be improved to make the code more readable:

- `totalSupplyAmount` change to `totalSupply`
- `assetContract` change to `asset`
- `decimalsValue` change to `decimals`
- `sharesName` change to `name`
- `sharesSymbol` change to `symbol`
- `sharesBalanceOf` change to `balanceOf`
- `sharesAllowance` change to `allowance`

Recommendation

We recommend changing the names of variables to improve code readability and observe naming compatibility with ERC20 standard for the vault tokens.

Update

Client's response

We need to keep our names as it is because we are inheriting IERC20, IERC20Metadata and IERC4626 and we need to override some functions that has the exactly names that they are

suggesting us to change which would lead us to have a lot of TypeError: Contract "VaultPackage" should be marked as abstract. errors if we try to change

I-07	Redundant `_onlySelf` function in `BaseStrategy`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
BaseStrategy.sol	contract `BaseStrategy` > function `_onlySelf`	80

Description

The function `_onlySelf` of the contract `BaseStrategy` is solely utilized once in `onlySelf` modifier. With an `internal` visibility modifier, it cannot be invoked from outside the contract. Furthermore, its implementation consists of just one line:

```
function _onlySelf() internal view {
  require(msg.sender == address(this), "!self");
}
```

Recommendation

We recommend streamlining code by moving the single line implementation of the `_onlySelf` function into the `onlySelf` modifier, thereby eliminating the need for the redundant function.

Update

Client's response

Redundant fuction was removed in commit `089c7c823f5d2763034db378470901c16283ebaf`.

I-08	No withdraw stuck tokens functionality in `TokenizedStrategy`, `VaultPackage`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
TokenizedStrategy.sol	contract `TokenizedStrategy`	31
VaultPackage.sol	contract `VaultPackage`	24

Description

In the mentioned contracts, there is no functionality to withdraw stuck tokens. As these contracts are public and used by third parties, it is probable that some tokens are sent to them by mistake. Therefore, a functionality to withdraw stuck tokens is needed.

Recommendation

We recommend implementing functionality to withdraw stuck tokens for these contracts.

Update

Client's response

Fixed on commit `15a5c0c52baca13c58950b65036465a4db4c2bd6`

I-09	Missing check that `tend` logic is implemented in `TokenizedStrategy`
Severity	INFO
Status	• NO ISSUE

Location

File	Location	Line
TokenizedStrategy.sol	contract `TokenizedStrategy` > function `tend`	496

Description

In the function `tend` of contract `TokenizedStrategy`, there is no check to verify that the tend logic is implemented in the `BaseStrategy` contract. As a result, if the tend logic is not implemented, calling the tend function may lead to an unpredictable revert without a clear error reason.

Recommendation

We recommend incorporating `tendTrigger` function of the `BaseStrategy` contract to ensure that the tend logic is implemented before executing the tend function.

Update

Client's response

Tend logic is optional and having no logic by default won't lead to any unpredictable revert.

I-10	Unused function in `VaultPackage`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
VaultPackage.sol	contract `VaultPackage` > function `_isContract`	1732

Description

The function `_isContract` of the contract `VaultPackage` isn't utilized anywhere in the project codebase.

Recommendation

We suggest removing the unused function to enhance code readability and reduce the gas cost of contract deployment.

Update

Client's response

Unused function was removed in commit `089c7c823f5d2763034db378470901c16283ebaf`.

I-11	`Int` type initialization to zero is redundant in `VaultPackage`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
VaultPackage.sol	contract `VaultPackage` > function `_manageUnlockingOfShares`	932
VaultPackage.sol	contract `VaultPackage` > function `_issueSharesForAmount`	1057
VaultPackage.sol	contract `VaultPackage` > function `_maxWithdraw`	1493
VaultPackage.sol	contract `VaultPackage` > function `_assessProfitAndLoss`	1572
VaultPackage.sol	contract `VaultPackage` > function `_assessProfitAndLoss`	1573
VaultPackage.sol	contract `VaultPackage` > function `_unlockedShares`	1594

Description

In the mentioned locations variables are initialized to zero. These initializations are redundant because zero is the default value of `int/uint` type variable in Solidity.

Recommendation

We recommend removing redundant initialization to zero.

Update

Client's response

The issue is fixed in the following lines:
`089c7c823f5d2763034db378470901c16283ebaf#L1616`
`089c7c823f5d2763034db378470901c16283ebaf#L1594-R1588`
`089c7c823f5d2763034db378470901c16283ebaf#L1515-R1508`
`089c7c823f5d2763034db378470901c16283ebaf#L1415-R1408`
`089c7c823f5d2763034db378470901c16283ebaf#L1326-R1319`
`089c7c823f5d2763034db378470901c16283ebaf#L1089-R1080`
`089c7c823f5d2763034db378470901c16283ebaf#L964-R957`

I-12	Redundant inheritance from the contract `ReentrancyGuard` in `FactoryStorage`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
FactoryStorage.sol	contract `FactoryStorage`	10

Description

In the contract `FactoryStorage`, there is inheritance from the `ReentrancyGuard` contract, yet its functionality remains unused.

Recommendation

We recommend eliminating the redundant inheritance from the `ReentrancyGuard` contract to enhance optimization and maintain codebase cleanliness.

Update

Client's response

The issue was fixed in commit `089c7c823f5d2763034db378470901c16283ebaf`.

I-13	Possible to erroneously set a very long distribution period in `Investor`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
Investor.sol	contract `Investor` > function `setupDistribution`	67

Description

In the function `setupDistribution` of the `Investor` contract, there are no restrictions on setting an arbitrarily large value for `distributionEnd`. This creates the risk of accidentally configuring an exceptionally long distribution period during a function call, and once set, it cannot be altered.

Recommendation

We recommend adding limits on the duration of the distribution period to prevent inadvertent and excessively long configurations.

Update

Client's response

The issue was fixed in commit `089c7c823f5d2763034db378470901c16283ebaf`.

I-14	Redundant call to `_maxDeposit` function in `VaultPackage`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
VaultPackage.sol	contract `VaultPackage` > function `_deposit`	1088
VaultPackage.sol	contract `VaultPackage` > function `_mint`	1123

Description

In the mentioned locations, there is a redundant possibility of calling the `_maxDeposit` function twice:

```
if (assets > _maxDeposit(recipient)) {
    revert ExceedDepositLimit(_maxDeposit(recipient));
}
```

Recommendation

We recommend considering the possibility of calling the `_maxDeposit` function once, storing the result in memory to avoid redundant calls within the `if` block.

Update

Client's response

The issue was fixed in the following lines:
089c7c823f5d2763034db378470901c16283ebaf#L1160-R1154
089c7c823f5d2763034db378470901c16283ebaf#L1160-R1154

I-15	Checking `assets` for zero occurs after calling `_maxDeposit` in `VaultPackage`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
VaultPackage.sol	contract `VaultPackage` > function `_deposit`	1091

Description

In the function `_deposit` of contract `VaultPackage`, the maximum value for `assets` is checked first using the `_maxDeposit` function, followed by a less resource-intensive check for `0`:

```
if (assets > _maxDeposit(recipient)) {
    revert ExceedDepositLimit(_maxDeposit(recipient));
}
if (assets == 0) {
    revert ZeroValue();
}
```

Recommendation

We recommend swapping the checks for `0` and for the maximum value to optimize gas consumption.

Update

Client's response

The issue was fixed in commit `089c7c823f5d2763034db378470901c16283ebaf`.

I-16	Unnecessary prohibition on setting `depositLimit` equal to zero in `VaultPackage`
Severity	INFO
Status	• ACKNOWLEDGED

Location

File	Location	Line
VaultPackage.sol	contract `VaultPackage` > function `setDepositLimit`	116

Description

In the function `setDepositLimit` of contract `VaultPackage`, setting `_depositLimit` equal to `0` is prohibited:

```
if (_depositLimit == 0) {
    revert ZeroValue();
}
```

This restriction is unnecessary because if `depositLimit` is set to `1`, the logic check in the `_maxDeposit` function will also reject only deposits equal to `0`:

```
uint256 currentDepositLimit = depositLimit;
if (currentTotalAssets >= currentDepositLimit) {
    return 0;
}
```

Recommendation

We recommend removing the check for deposit limit equality to zero.

Update

Client's response

I agree that 1 would have the same effect as 0, but 0 is can be easily set by mistake since it's default value for uint. I mean, passing 0 value is a common issue, and at least we can handle it here. and the fact that we can't handle all unexpected values doesn't mean we shouldn't handle any.

I-17	Increment to empty value in `VaultPackage`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
VaultPackage.sol	contract `VaultPackage` > function `_calculateShareManagement`	1449

Description

In the function `_calculateShareManagement` of contract `VaultPackage`, the `+=` operator is used when no value has been assigned to `shares.sharesToBurn` yet:

```
shares.sharesToBurn += _convertToShares(loss + totalFees, Rounding.ROUND_UP);
```

Recommendation

We recommend replacing `+=` with the assignment operator.

Update

Client's response

The issue was fixed in commit `089c7c823f5d2763034db378470901c16283ebaf`.

I-18	No event about setting `vaultPackage` in `FactoryPackage`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
FactoryPackage.sol	contract `FactoryPackage` > function `initialize`	30

Description

In the function `initialize` of contract `FactoryPackage`, values are set for `vaultPackage`, `feeRecipient` and `feeBPS`, but only the event for changing the fee is emitted:

```
emit FeeConfigUpdated(_feeRecipient, _feeBPS);
```

Recommendation

We recommend also adding an emission for the `VaultPackageUpdated` event.

Update

Client's response

The issue was fixed in commit `089c7c823f5d2763034db378470901c16283ebaf`.

I-19	No validation of the same `vaultPackage` address during installation in `FactoryPackage`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
FactoryPackage.sol	contract `FactoryPackage` > function `updateVaultPackage`	35

Description

In the function `updateVaultPackage` of contract `FactoryPackage`, there is no validation to prevent setting the same address again. This results in emitting an event without actually changing the `vaultPackage` itself.

Recommendation

We recommend adding a validation to check whether the `vaultPackage` has changed before emitting the event.

Update

Client's response

The Issue was fixed in commit `089c7c823f5d2763034db378470901c16283ebaf`.

I-20	Immutable variable in the section for constants in `BaseStrategy`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
BaseStrategy.sol	contract `BaseStrategy`	101

Description

In the contract `BaseStrategy`, there is a section for constants, but it contains only one variable, which is immutable:

```
/*////////////////////////////////////////////////////////////////
                        CONSTANTS
////////////////////////////////////////////////////////////////*/
// ...
address public immutable tokenizedStrategyAddress;
```

Recommendation

We recommend moving the variable to the section for immutable variables to avoid confusion.

Update

Client's response

Fixed in commit `089c7c823f5d2763034db378470901c16283ebaf`.

4.5 VULNERABILITIES AND MITIGATIONS

Throughout the audit process, the team identified several potential vulnerabilities, aligning with the common attack vectors highlighted in the introduction part of the report. The majority of these vulnerabilities are associated with smart contract bugs, which could potentially expose the protocol to various security risks. These vulnerabilities include but are not limited to issues such as math errors, front-running issues, input validation shortcomings, and logic flow vulnerabilities.

Researched Attack Vectors

- **Flash Loan Attacks**
  no vulnerabilities found
- **Reentrancy Attacks**
  no vulnerabilities found
- **Liquidity Pool Exploitation**
  not applicable
- **Smart Contract Bugs**
  vulnerabilities reported
- **Front-Running**
  vulnerabilities reported
- **Supply Chain Attacks**
  no vulnerabilities found
- **Access Control Attacks**
  no vulnerabilities found

5 CONCLUSION

The vulnerabilities found have been fixed and part of it acknowledged by the team, fixes are planned in the next updates.

The following table contains the total number of issues that were found during audit:

Severity	FIXED	ACKNOWLEDGED	NO ISSUE	Total
CRITICAL	1	0	0	1
MAJOR	0	1	0	1
WARNING	5	0	0	5
INFO	16	2	2	20
Total	22	3	2	27

THANK YOU FOR CHOOSING