OpenZeppelin | security

OpenZeppelin ERC4626 Tokenized Vault Audit

November 15th, 2022

This security assessment was prepared by OpenZeppelin.

Table of Contents

Summary

Type	Library
Timeline	From 2022-10-16 To 2022-10-28
Languages	Solidity

Total Issues	8 (2 resolved, 1 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	1 (0 resolved)
Medium Severity Issues	0 (0 resolved)
Low Severity Issues	2 (1 resolved)
Notes & Additional Information	5 (1 resolved, 1 partially resolved)

Scope

We audited the `OpenZeppelin/openzeppelin-contracts` repository at the `14f98dbb581a5365ce3f0c50bd850e499c554f72` commit.

In scope were the following contracts:

```
contracts
├── token
│   └── ERC20
│       └── extensions
│           └── ERC4626.sol
└── utils
    ├── math
    │   └── Math.sol
```

Introduction

The OpenZeppelin Contracts team asked us to review their new contract implementation of EIP4626, namely, the ERC4626 contract.

EIP4626 aims to define a standard way to represent an ERC20 token through shares of a vault contract. The vault is itself an ERC20 token and its quantities are called *shares*, while the underlying token amounts that shares represent are called *assets*. Through the minting / burning of shares the vault manages the deposit / withdrawal of assets according to specific conversion rates. Moreover, specific functions are defined to preview the expected amount of shares / assets by a deposit / withdrawal operation. Users might decide to directly deposit an amount of assets or mint an exact amount of shares, and conversely they can decide to withdraw an amount of assets or redeem an exact amount of shares. How many shares correspond to a given amount of assets is determined by conversion rates, from asset to shares and vice versa.

The OpenZeppelin implementation is a vanilla implementation of this EIP where conversion rates are simply the relative quantities of both assets and shares present in the vault.

The EIP also defines optional fees and slippage mechanism which are not implemented in the OpenZeppelin contract.

The contract is meant to be a general purpose one. Many of the core functions are overridable and there are opportunities for developers to implement custom functionalities on top of it. Because of this, there are some dangerous pitfalls that may not directly affect the current implementation but that should be correctly brought up to developers trying to come up with a custom implementation that makes use of this contract.

- There's an invariant that is controlled by the `_isVaultCollateralized` function. This is stopping users from depositing when `totalSupply > 0` and `totalAssets = 0` because, in that case, any amount of `assets` being deposited would represent an infinite amount of shares. However, this applies only to `deposit` but not to `mint`. This means that if the invariant is broken, users can still mint the maximum number of shares for `0` amount of assets. Currently, the invariant is unlikely to be broken but one must consider that `ERC4626` is abstract and must be implemented in a custom made contract. This eventuality must be correctly presented to unaware developers approaching this contract.

- The underlying asset balance of the vault is used to calculate the ratio of shares to assets for issuance and redemption. It is important that users understand this, so that they are very careful if the vault allows transfers of underlying assets for more than one block. This would imply that there could exist a time window in which the underlying asset of the vault is out of the vault, and that a malicious user can mint shares at below true cost, diluting all positions of other existing users. For the same reason, vaults should also not approve spenders for the underlying asset. Similarly, if the underlying asset is moved out without accounting for such temporary or permanent decrease through a `burn` of shares, malicious actors can artificially dilute every user that holds some shares at little to no cost. Notice that rebasing tokens might have a similar effect and any use of them should be correctly analyzed in the context of the vault mechanism.

- The `ERC4626` tokens derived from the vault might be generally added or listed to other decentralized protocols. This means that any integrations require prior understanding of the tradeoffs of this token contract when it comes to interacting with other contracts. One specific topic that must be taken into account is if `ERC4626` tokens are made flash-loanable. In that case, the `withdraw` and `redeem` functions might suffer from an inflation attack where the `owner` can inflate his own balance to manipulate the outcome of the `withdraw` and `redeem` calls. Specific scenarios where this might be troublesome, were not identified given the fact that `withdraw` and `redeem` will burn owner's tokens. However, protocols that allow flash-loan repays with different assets, or whales in general, might still use artificial inflations for yet-unknown purposes.

The commit used for the audit is 14f98db and the code has been audited during the course of 5 days by two auditors. Here we present our findings.

Findings

High Severity

H-01 Vault deposits can be front-run and user funds stolen

The current implementation of the `ERC4626` contract is susceptible to an underlying asset balance manipulation attack. When a user performs a `deposit`, the amount of shares they will receive is calculated by taking the number of assets provided, multiplying that by the number of existing shares in circulation, and then dividing the result by all assets owned by the vault. The result is then rounded down, following the EIP specifications. The problem arises when the value of `totalAssets` is manipulated to induce unfavorable rounding for new users depositing into the vault at the benefit of users who already own shares of the vault. The most extreme example of this attack is when a user is the first to enter the vault. In the scenarios that both the shares and underlying assets of a vault use `18` decimals:

- The first user deposits a single underlying asset in exchange for a single share of the vault.
- Both the `supply` and `totalAssets` are `1` during the calculation for the next deposit.
- An innocent user attempts to deposit `1e18` tokens into the vault, expecting `1e18` shares in return.
- The first user front runs the innocent user's transaction with a transaction that directly transfers `1e18` of the underlying asset to the vault
- The vault's `supply` remains `1` while the `totalAssets` is now in fact `1e18 + 1`
- The innocent user's deposit call is processed and the amount of shares received is calculated as `1 * 1e18 / (1e18 + 1)` which rounds down to `0`.
- The first user now owns `1` share that is worth `2e18 + 1` of the underlying asset and the second user is out their deposit.

In this scenario, the first user may be able to continue this attack in perpetuity, draining the funds of any user who attempts to deposit into the vault.

While solving this attack entirely without creating an overly restrictive library is non-trivial, consider taking steps to reduce the severity of this issue. One such step would be to enforce that `0` shares will never be minted. While it is true that this only partially solves the problem, because the underlying balance can still be manipulated to impact rounding (i.e. a user should have been minted `2` shares but with the balance manipulation attack that became `1.99`

shares which is rounded down to `1` share), this reduces the risk of a user draining all funds in perpetuity.

Without the ability to prevent other users from joining the vault, the balance manipulation attack quickly becomes economically infeasible to continue. Another item to consider and that is mentioned in the contract docstrings would be to build out an extension of the `ERC4626` contract that has protections built-in for standard vault use cases.

We understand that the contract is meant to be a general purpose contract that should be correctly implemented by developers. However, if actions in limiting such unwanted scenarios are not actively pursued, consider refactoring the docstrings to be as clear as possible about all potential outcomes and all the possible solutions, given that the current docstrings don't highlight the issue in its entirety.

Update: *Acknowledged, not resolved. The OpenZeppelin team will implement mitigations for this issue. Current discussions are happening here and once the final solution is confirmed, the team will update the contracts accordingly.*

Low Severity

L-01 Unclear function

The `_isVaultCollateralized` function of the `ERC4626` contract is used to guarantee deposits cannot be made in the unwanted scenario where the vault's balance of the underlying token is zero and the total supply of minted shares is more than zero.

In this scenario, shares are worth zero underlying assets and because of this, any asset deposited would correspond to an infinite number of shares. Even though it would not be possible to deposit under these circumstances because the `mulDiv` function would revert, the team decided to add this function to follow EIP specifications and to assure that the problem is noticed early and in a clear way.

However, the name of the function is inconvenient, since a **collateralized** status often refers to an asset balance being lower or equal to that of the underlying asset. Even if the function is checking that the underlying asset balance is greater than zero, this is a necessary but insufficient condition to determine collateralized status as the vault's shares balance is not considered in this check.

Moreover, the function returns true even in the case that both the shares and the underlying balances are zero, but in this case, the vault state is more properly described as in a **null** or **uninitialized** state instead of a **collateralized** one.

Notice that the `mint` function has the same effect of minting new shares and depositing assets inside the contract, but it doesn't make use of the `_isVaultCollateralized` function. In the scenario where shares are worth zero assets, users can therefore `mint` shares for free. Whether this scenario is to be avoided or allowed depends on the final implementation of the vault but should be properly documented.

Since the function is meant to alert the depositor of the health status of the vault, consider renaming this function to something more appropriate. Some suggestions might be:

- `_isVaultHealthy`
- `_isShareWorthZero`
- `_areSharesWorthless`

Update: *Resolved in commit b189e6a. Moreover, docstrings into the `mint` functions have been added in commit 8bc307f.*

L-02 Unsafe ABI encoding

It is not an uncommon practice to use `abi.encodeWithSignature` or `abi.encodeWithSelector` to generate call data for a low-level call. However, the first option is not typo-safe and the second option is not type-safe. The result is that both of these methods are error prone and should be considered unsafe.

Throughout the `ERC4626` and, in general, the entire OpenZeppelin contracts library, there are several occurrences of unsafe ABI encodings being used. The `encodeCall` functionality requires a Solidity version greater or equal than the 0.8.11 and this might be a breaking change for developers working previous versions. However, in future versions, we strongly recommend to the team to take into consideration replacing all the occurrences of unsafe ABI encodings with `abi.encodeCall`, which checks whether the supplied values actually match the types expected by the called function and also avoids errors caused by typos.

Update: *Acknowledged, not resolved. The OpenZeppelin team stated:*

> We plan to make this change in the next major version. As mentioned in the report, we can't adopt it in the current version due to backwards compatibility. For reference, check #3693.

Notes & Additional Information

N-01 EIP inconsistency

The `ERC4626` contract intends to implement a tokenized vault contract that strictly adheres to EIP-4626 and its definitions. However we detected some mismatches within the use of named and unnamed return parameters. Specifically, the `asset`, `totalAssets`, `mint`, `deposit`, `withdraw` and `redeem` functions return unnamed parameters while the EIP defines named return parameters for these functions.

In order to perfectly match the EIP standard and follow its definition, consider refactoring the code and fixing the discrepancies.

Update: *Acknowledged, not resolved. The OpenZeppelin team stated:*

> We consider variable names in EIPs as non-normative guidelines, and we occasionally do things differently to keep consistency within the library. Return parameters in particular are not named throughout the library, save for a few exceptions.

N-02 Inconsistent formatting

The `Deposit` and `Withdraw` events are similar in length and location, but they are presented differently in the `IERC4626` contract. Consider matching the formatting for these two events to improve consistency and readability within the codebase.

Update: *Acknowledged, not resolved. The OpenZeppelin team stated:*

> Formatting is done automatically by Prettier, and in this case one of the events is over the line length limit while the other isn't. In this case there is no way to make formatting consistent while using the autoformatter.

N-03 Missing docstrings

Throughout the `ERC4626` contract, there are several parts that do not have docstrings. For instance:

- The contract docstrings are missing information on the functions and actions needed for the abstract contract to be implemented. It should be pointed out that the constructor of the `ERC20` contract, from which it extends, should be called in the implementation contract, defining `name` and `symbol`.
- The `_isVaultCollateralized` function is missing any sort of docstring.

In order to improve clarity and understandability, consider improving the contract docstrings.

Update: *Partially resolved in commit 12fa301. The first reported case has not been resolved. For that, the OpenZeppelin team stated:*

> *The suggestion about contract-level docstrings was not addressed because the pattern of inheriting a contract but not calling the constructor is used widely throughout the library. It isn't specific to this contract, so addressing it requires a larger effort to see how we communicate it in the documentation, but we do note that so far it hasn't been an issue for our users (we never get support requests about it).*

N-04 Missing error message in require statement

Within `Math.sol` there is a `require` statement on line 78 that lacks an error message.

Consider including specific, informative error messages in `require` statements to improve overall code clarity and to facilitate troubleshooting whenever a requirement is not satisfied.

Update: *Resolved in commit 827c8cc.*

N-05 Non-explicit imports are used

Non-explicit imports are used inside the `ERC4626` contract, which reduces code readability and could lead to conflicts between names defined locally and the ones imported. This is especially important if many contracts are defined within the same Solidity files or the inheritance chains are long.

We know that the OpenZeppelin contracts library has not leveraged explicit, named imports to date which would create inconsistencies with what has been done in the past. However,

following the principle that clearer code is better code, consider using named import syntax (`import {A, B, C} from "X"`) to explicitly declare which contracts are being imported.

Update: *Acknowledged, not resolved. The OpenZeppelin team stated:*

> We will consider this for the next major version as this is a relatively large change and potentially breaking if users are assuming something to be indirectly in scope.

Conclusions

One (1) high and one (1) medium severity issues were found along with a couple of low severity issues and notes. Recommendations have been given to improve the current state of the codebase. We are happy to see the OpenZeppelin Contract team going the extra mile in implementing EIP-4626 with complex designs in an effort to provide the community with a ready and safe to use implementation that can be adopted as it is out of the box.

However, given the complexities and the potential issues that can arise from an incorrect custom implementation of such a contract, we strongly suggest improving and raising more awareness in the docstrings, with the final goal of correctly educating the community toward potential dangers that can result from it's incorrect use.