PUBLIC

Code Assessment

of the USDO Smart Contracts

August 12, 2024

Produced for

OpenEden

by

CHAINSECURITY

Contents

1 Executive Summary

Dear OpenEden team,

Thank you for trusting us to help OpenEden with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of USDO according to Scope to support you in forming an opinion on their security risks.

OpenEden implements a USD stablecoin called USDO and a wrapper contract (acting as a vault) for it called wUSDO. The stablecoin will be backed by yield-earning U.S. treasury bills.

We did not find any critical issues in the codebase. Yet, we recommended testing wUSDO deposits, transfers and withdrawals with intermediate multiplier changes in USDO very carefully. All raised issues were addressed accordingly.

In summary, we find that the codebase provides a high level of security as most critical operations are access-controlled and should have additional off-chain procedures to ensure the security of the system. However, it is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	4
• Risk Accepted	2
• Acknowledged	2

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the USDO repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	29 July 2024	50c3cb84f74ffdb5a1235e375fc3ef81c5f59aa2	Initial Version
2	12 August 2024	1c8addeb9bc32ed945c10287f7c673543b900be6	After Intermediate Report

For the solidity smart contracts, the compiler version `0.08.18` was chosen, and the following two contracts were reviewed:

- `USDO.sol`
- `wUSDO.sol`

2.1.1 Excluded from scope

Excluded from the scope are all third-party libraries imported into the contracts.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

OpenEden offers a yield-bearing stablecoin system that consists of two contracts: USDO and wUSDO. USDO is a rebasing ERC-20 token that accrues interest over time and is minted and burned by a central party. wUSDO is a non-rebasing, ERC-4626 compliant wrapper contract for USDO.

2.3 Protocol

2.3.1 USDO

The `USDO` token contract is an upgradable ERC-20 token with rebasing balances. It inherits the following default OpenZeppelin contracts:

- `AccessControlUpgradeable`: contract that gives role-based access control to admin functions.
- `PausableUpgradeable`: contract that allows the contract to be paused and unpaused.
- `EIP712Upgradeable`: contract that allows hashing and signing of structured data.

- `UUPSUpgradeable`: contract that allows to upgrade the logic of the implementation.

The balances of the contract increase and decrease with a so-called `bonusMultiplier` that controls the conversion rate between shares and token balances. The multiplier is initially set to 1e18 during initialization and can be incremented by the `MULTIPLIER_ROLE`. The contract stores the shares owned by each user and the user's balance is then calculated as the number of shares multiplied by the current value of the bonusMultiplier:

$$balance = shares * bonusMultiplier / 1e18$$

The `DEFAULT_ADMIN_ROLE` can re-set the `bonusMultiplier` to any value >=1e18 and can thereby reduce the value of the bonusMultiplier and the value of the shares.

Addresses with the `MINTER_ROLE` can mint new tokens for users and user addresses with the `BURNER_ROLE` can burn tokens. Minting tokens takes as an argument the number of tokens to mint, and the contract then calculates the number of shares to mint based on the current value of the `bonusMultiplier`.

$$shares = balance * 1e18 / bonusMultiplier$$

A privileged address with the `BURNER_ROLE` can burn tokens from user accounts. This reverses the operation from above.

The contract can be paused by the `PAUSE_ROLE`. When paused, no transfers are allowed, including token burning and minting.

The `BANLIST_ROLE` can ban and un-ban addresses. Banning stops users from transferring any tokens from their addresses. This includes burning tokens from a banned address (burning is a transfer to address 0).

The so-called `UPGRADE_ROLE` can upgrade the implementation of the contract. This is done by calling the `upgradeTo` function of the `UUPSUpgradeable` contract.

In addition to the standard ERC-20 function, the contract implements EIP-2612 permit, to allow users to approve token transfers without having to send a transaction to the blockchain and `increaseAllowance` and `decreaseAllowance` functions to protect users against frontrunning approvals. Note that the latter are taken from the OpenZeppelin 4.9 release and are not part of the current OpenZeppelin release.

2.3.2 wUSDO

The `wUSDO` token contract is a non-rebasing ERC-4626 compliant wrapper contract around the rebasing `USDO` token.

They inherit the same default OpenZeppelin contracts as the `USDO` token:

- `AccessControlUpgradeable`: contract that gives role-based access control to admin functions.
- `PausableUpgradeable`: contract that allows the contract to be paused and unpaused.
- `EIP712Upgradeable`: contract that allows hashing and signing of structured data (used for ERC20-permit).
- `UUPSUpgradeable`: contract that allows to upgrade the logic of the implementation.

and in addition, inherit the ERC-4626 Vault contract implementation by OpenZeppelin:

- `ERC4626Upgradeable` contract that implements the ERC-4626 Vault contract.

Users can deposit USDO tokens into the wUSDO contract to receive wUSDO tokens. These tokens are non-rebasing and can be utilized in external DeFi applications. Users can redeem their wUSDO tokens for USDO. The conversion between shares is handled by OpenZeppelin's ERC-4626 Vault contract.

The contract has a dedicated `PAUSE_ROLE` to pause and unpause the contract. When paused, no transfers are allowed. Additionally, the contract integrates the pause function from the USDO contract to

stop transfers when the underlying contract is paused. Similarly, it integrates the banlist from the USDO contract to halt transfers when the user is banned in USDO. This means that banning a user will prevent them from transferring any tokens.

The contract has a `UPGRADE_ROLE` that can upgrade the implementation of the contract by calling the `UUPSUpgradeable.upgradeTo()`.

In addition to the standard ERC-20 function, the contract implements EIP-2612 permit, to allow users to approve token transfers without having to send a transaction to the blockchain.

2.4 Trust Model

The admin roles that are described above are fully trusted and are expected to behave in the interest of the users. The most powerful role is the default admin which can transfer all other roles to other accounts. They are expected to only transfer roles to well-vetted addresses.

Another important role is the `UPGRADE_ROLE` role that can upgrade the implementation of the contracts. They are expected to upgrade the contract to well-tested and non-malicious implementations.

2.5 Assumptions

- We assume that U.S. treasury bills do not default fully or partially. Furthermore, we assume that the rate does not decrease in such a way that it causes issues in the system (e.g. to fall below `_BASE` in the beginning).
- We assume all mint as well as burn operations are carefully checked. If burn operations are automatic and instant, front-running issues might arise because rate changes will be visible in the mem-pool before they are applied.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	4

5.1 Dust Accumulation in wUSDO

Correctness Low Version 1 Risk Accepted

CS-USDOO-001

Through the back-and-forth conversion from assets to shares in USDO, rounding errors will accumulate. This prevents the full withdrawal of all USDO from wUSDO. Over time, dust will accumulate in wUSDO with no owner. This will also marginally influence the shares received when depositing in wUSDO. We recommend testing wUSDO deposits, transfers and withdrawals with intermediate multiplier changes in USDO very carefully.

Risk accepted:

OpenEden has accepted the risk and states that they will add additional tests to cover more edge cases.

5.2 Minting to Banned Addresses

Design Low Version 1 Acknowledged

CS-USDOO-002

Banning an address disallows the address to send funds to other addresses and to burn the address's funds. However, it remains possible to mint funds to the banned address.

Acknowledged:

OpenEden stated:

```
The code won't check `to` for gas efficiency.
```

5.3 Redeem Without Reducing Shares

Design | Low | Version 1 | Risk Accepted

CS-USDOO-003

In function `USDO.burn()` the amount of shares burned is rounded to zero when `(amount * _BASE) < bonusMultiplier` (where `amount > 0`). This could cause problems when a user redeems their USDO for USDC and the respective `amount` of USDO is burned. In this scenario, `_burn` will remove no shares and `_shares[user]` remain constant, while the user receives USDC. While this condition can be enforced in some external logic that is not part of the reviewed codebase, the issue should ideally be prevented and checked on a smart contract level.

Risk accepted:

The OpenEden has accepted the risk and noted:

```
The burn() is fully controlled by the admin, and in reality this won't happen.
```

5.4 Token Transfers Allowed to Token Contract Addresses

Design | Low | Version 1 | Acknowledged

CS-USDOO-004

The contracts allow sending tokens to the issuing contract (e.g., USDO token to USDO contract). Currently, there is no function to recover the funds. There might be the possibility to add this functionality in an update. However, we do not see a reason to allow this functionality.

Acknowledged:

OpenEden team has acknowledged the issue, but has decided to keep the code unchanged.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0
Informational Findings	2

- Gas Optimizations Code Corrected
- Arguments Returned by Custom Error Code Corrected

6.1 Arguments Returned by Custom Error

Informational Version 1 Code Corrected

CS-USDOO-009

The function `USDO.permit()` reverts with custom error `ERC2612InvalidSignature()` that returns the owner and spender addresses.

```
address signer = ECDSAUpgradeable.recover(hash, v, r, s);

if (signer != owner) {
    revert ERC2612InvalidSignature(owner, spender);
}
```

However, it does not return the singer of the signature, which is commonly done by current libraries like OpenZeppelin.

Code corrected:

The OpenEden repository has been updated to return the `signer` and `owner` addresses in the custom error:

```
if (signer != owner) {
    revert ERC2612InvalidSignature(signer, owner);
}
```

6.2 Gas Optimizations

Informational Version 1 Code Corrected

CS-USDOO-008

1. In the function `USDO._updateBonusMultiplier()`, the event `BonusMultiplier` can log the function argument `_bonusMultiplier` instead of reading `bonusMultiplier` from storage.
2. The mapping `_bannedList` could use integer values to indicate true or false. As all words are unit256 by default, using boolean will add a type conversion from uint to bool. The gas for this casting could be saved.

Code corrected:

The optimizations have been implemented in Version 2.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Decreasing BonusMultiplier

Informational **Version 1** **Acknowledged**

CS-USDOO-005

We assumed that the `BonusMultiplier` is never decreasing. This assumption is not true if market prices are factored into the multiplier instead of interest payments only. In this case, the chances are high that the multiplier would need to be decreased. However, we assumed that the market prices are not factored into the multiplier.

Technically, it is possible to decrease the `BonusMultiplier` by calling the `setBonusMultiplier` function. However, a decrease in the beginning to a value below `_BASE` will not be possible due to the restriction `_bonusMultiplier < _BASE`. Additionally, `setBonusMultiplier` allows setting the same multiplier multiple times.

Another remaining scenario where the `BonusMultiplier` would need to decrease is partial or full treasury bill defaults. This scenario is highly unlikely but should be considered in the design. Otherwise, these scenarios need to be properly handled off-chain as the smart contract accounting could be incorrect.

Acknowledged:

OpenEden stated:

> `Won’t happen in reality, and can upgrade if needed.`

7.2 Shares to Burn Are Rounded Down

Informational **Version 1** **Acknowledged**

CS-USDOO-006

The function `USDO._burn()` rounds down the number of shares burned from the user.

```
function _burn(address account, uint256 amount) private {
    ...

    uint256 shares = convertToShares(amount);
    uint256 accountShares = sharesOf(account);
```

Note, this is in contrast to the natspec above function `USDO.convertToShares()` which states:

```
 * Note: All rounding errors should be rounded down in the interest of the protocol's safety.
 * Token transfers, including mint and burn operations, may require a rounding, leading to potential
 * transferring at most one GWEI less than expected aggregated over a long period of time.
```

```
 */
function convertToShares(uint256 amount) public view returns (uint256) {
    return (amount * _BASE) / bonusMultiplier; // amount / index = shares
}
```

However, this remains an informational issue, since the issuer has full control over how much USDC to send back to the user and can always send a bit less to compensate for the rounding down.

Acknowledged:

OpenEden team has acknowledged the issue, but has decided to keep the code and specification unchanged.

7.3 Upper Bound for Bonus Multiplier

Informational **Version 1** **Acknowledged**

CS-USDOO-007

The function `USDO._updateBonusMultiplier()` enforces that the new value of the bonus multiplier is greater than, or equal to, 1e18. However, there is no upper bound for the bonus multiplier.

Limiting the increase of the bonus multiplier might prevent unintentional high increases (e.g., caused by an error). Currently, the annual increase could e.g., be 100% (i.e. 1e18). This would not make sense for a bond value that is expected to be updated daily. The function `USDO.addBonusMultiplier()` could revert if `_bonusMultiplierIncrement` is greater than a certain threshold e.g., 3e15 (1e18 / 365 days = 2.7e15).

Acknowledged:

OpenEden acknowledges the issue and stated that they will monitor changes in the bonus multiplier off-chain.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Effects of Rounding in USDO

Note **Version 1**

In each state-modifying functions the token amount is taken as argument and converted to the share representation by rounding down:

```
function convertToShares(uint256 amount) public view returns (uint256) {
    return (amount * _BASE) / bonusMultiplier;
}
```

The rounding has the following effects:

- When minting tokens, minting of `X` tokens will effectively mint `Y` tokens, where $Y \leq X$. Similarly, burning of `X` tokens will effectively burn `Y` tokens, where $Y \leq X$.
- For transfers of `X` tokens with initial balances `A` and `B`, the updated balances `A'` and `B'` satisfy $A \geq A - X$ and $B' \leq B + X$.
- The total supply invariant is not strictly enforced:

$$\sum_{i \in holder} balanceOf(i) \leq totalSupply$$

Note that the functions `transfer()`, `transferFrom()`, `mint()` and `burn()` will update the balances with amounts that might be different from those specified by callers due to this rounding errors. 3rd-party protocols should consider this when integrating with USDO.

8.2 Potentially Ineffective Banning

Note **Version 1**

The admin can ban an address. This address can then no longer send funds to other addresses. This address's funds also not be burned anymore if it is banned. But funds could be minted to a banned address as mentioned in Minting to banned addresses. If OpenEden does not use private mem-pools, the transaction will be publicly visible in the mem-pool before it is executed. Hence, the to-be-banned address can send the funds to another address before the ban is effective and escape the ban each time by successfully front-running the ban transaction.

8.3 To Address Not Checked

Note **Version 1**

As the comment in the code implies, it is intended that the current code base does not validate the `to` address in `_beforeTokenTransfer`. Consequently, it is possible to send funds to blocked addresses. These addresses will just not be able to send the funds to other addresses. However, it is still possible to mint to banned addresses (see Minting to banned addresses).