**** FOR INTERNAL USE ONLY ****

Ozean Finance Security Audit Report (Interim)

tags: `Ozean Finance`

1. INTRODUCTION

1.1 Disclaimer

The audit makes no statements or warranties about 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 fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only. The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of the Client. If you are not the intended recipient(s) of this document, please note that any disclosure, copying or dissemination of its content is strictly forbidden.

1.2 Security Assessment Methodology

A group of auditors are involved in the work on the audit. The security engineers check the provided source code independently of each other in accordance with the methodology described below:

1. Project architecture review:

- Project documentation review.
- General code review.
- Reverse research and study of the project architecture on the source code alone.

Stage goals

- Build an independent view of the project's architecture.
- Identifying logical flaws.

2. Checking the code in accordance with the vulnerabilities checklist:

- Manual code check for vulnerabilities listed on the Contractor's internal checklist. The Contractor's checklist is constantly updated based on the analysis of hacks, research, and audit of the clients' codes.
- Code check with the use of static analyzers (i.e Slither, Mythril, etc).

Stage goal

Eliminate typical vulnerabilities (e.g. reentrancy, gas limit, flash loan attacks etc.).

3. Checking the code for compliance with the desired security model:

- Detailed study of the project documentation.
- Examination of contracts tests.
- Examination of comments in code.
- Comparison of the desired model obtained during the study with the reversed view obtained during the blind audit.
- Exploits PoC development with the use of such programs as Brownie and Hardhat.

Stage goal

Detect inconsistencies with the desired model.

4. Consolidation of the auditors' interim reports into one:

- Cross check: each auditor reviews the reports of the others.
- Discussion of the issues found by the auditors.
- Issuance of an interim audit report.

Stage goals

- Double-check all the found issues to make sure they are relevant and the determined threat level is correct.
- Provide the Client with an interim report.

5. Bug fixing & re-audit:

- The Client either fixes the issues or provides comments on the issues found by the auditors. Feedback from the Customer must be received on every issue/bug so that the Contractor can assign them a status (either "fixed" or "acknowledged").
- Upon completion of the bug fixing, the auditors double-check each fix and assign it a specific status, providing a proof link to the fix.
- A re-audited report is issued.

Stage goals

- Verify the fixed code version with all the recommendations and its statuses.
- Provide the Client with a re-audited report.

6. Final code verification and issuance of a public audit report:

- The Customer deploys the re-audited source code on the mainnet.
- The Contractor verifies the deployed code with the re-audited version and checks them for compliance.
- If the versions of the code match, the Contractor issues a public audit report.

Stage goals

- Conduct the final check of the code deployed on the mainnet.
- Provide the Customer with a public audit report.

Finding Severity breakdown

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	**Description**
Critical	Bugs leading to assets theft, fund access locking, or any other loss of funds.
High	Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss funds.
Low	Bugs that do not have a significant immediate impact and could be easily fixed.

Based on the feedback received from the Customer regarding the list of findings discovered by the Contractor, they are assigned the following statuses:

Status	**Description**
Fixed	Recommended fixes have been made to the project code and no longer affect its security.

Status	Description
Acknowledged	The Customer is aware of the finding. Recommendations for the finding are planned to be resolved in the future.

1.3 Project Overview

The Ozean Finance protocol enables users to bridge stablecoins and migrate staked assets in a form of USDX from Ethereum to Ozean L2, where they can mint OzUSD, which is implemented following the ERC-4626 standard. The system employs strict security measures including allowlisting, deposit caps, and reentrancy protection.

1.4 Project Dashboard

Project Summary

Title	Description
Client	Ozean Finance
Project name	Ozean Finance
Timeline	10.12.2024 - 13.12.2024
Number of Auditors	3

Project Log

Date	Commit Hash	Note
10.12.2024	dacd9ed9c895c9b6be422531eea15fecc3c2b1d8	Commit for the audit

Project Scope

The audit covered the following files:

File name	**Link**
src/L2/WozUSD.sol	https://github.com/LayerLabs-app/Ozean-Contracts/blob/dacd9ed9c895c9b6be422531eea15fecc3c2b1d8/src/L2/WozUSD.sol
src/L2/OzUSD.sol	https://github.com/LayerLabs-app/Ozean-Contracts/blob/dacd9ed9c895c9b6be422531eea15fecc3c2b1d8/src/L2/OzUSD.sol
src/L1/USDXBridge.sol	https://github.com/LayerLabs-app/Ozean-Contracts/blob/dacd9ed9c895c9b6be422531eea15fecc3c2b1d8/src/L1/USDXBridge.sol
src/L1/LGEStaking.sol	https://github.com/LayerLabs-app/Ozean-Contracts/blob/dacd9ed9c895c9b6be422531eea15fecc3c2b1d8/src/L1/LGEStaking.sol
src/L1/LGEMigrationV1.sol	https://github.com/LayerLabs-app/Ozean-Contracts/blob/dacd9ed9c895c9b6be422531eea15fecc3c2b1d8/src/L1/LGEMigrationV1.sol

Deployments

File name	**Contract deployed on mainnet**	**Comment**

1.5 Summary of findings

Severity	**# of Findings**
CRITICAL	0
HIGH	0
MEDIUM	3
LOW	14

1.6 Conclusion

During the audit apart from checking well-known attack vectors and vectors that are presented in our internal checklist we carefully investigated the following attack vectors:

1. **Comprehensive Reentrancy Protection:** The protocol implements robust reentrancy protection across all token-handling functions through the `nonReentrant` modifier. This is complemented by a strict allowlist system that ensures only vetted tokens can interact with the protocol, preventing potential attacks through malicious token contracts. The `SafeERC20` library is consistently used for all token operations, providing additional protection against non-standard token implementations.

2. **Resilient State Management:** Changes to deposit caps and allowlists are handled through dedicated functions with proper access controls and event emissions. The system maintains consistent state even during parameter updates, with clear validation checks and no interdependencies that could lead to system-wide vulnerabilities. All state changes are atomic and properly validated.

3. **Secure Token Bridge Integration:** The bridging mechanism implements multiple security layers to ensure token integrity. The system enforces a strict 1:1 relationship between deposited tokens and minted USDX, with deposit caps and allowlist checks occurring before any token transfers. The architecture prevents unauthorized minting through mandatory validation of bridge transactions through the Optimism Portal and a proper decimal handling of deposited token and USDX.

4. **Secure Rebasing Implementation:** The OzUSD contract implements a shares-based rebasing mechanism with proper initialization safeguards. During deployment, a portion of initial shares is permanently locked by minting to a dead address (`0xdead`), establishing a secure foundation for the rebasing calculations. This approach ensures accurate share-to-token ratio calculations and prevents manipulation of the initial share price.

5. **L1 to L2 ETH Transfer Limitation:** The current implementation of the Optimism Stack architecture does not support direct ETH-to-USDX bridging on L2. This implementation enhances security by requiring all value transfers to go through the protocol's designated bridging contracts and supported stablecoin paths. The limitation prevents potential issues with 1:1 conversion of ETH to USDX during bridging.

6. **Read-Only Reentrancy Protection:** The OzUSD token implementation is inherently protected against read-only reentrancy attacks through its state management design. All balance-affecting operations are atomic and protected by the nonReentrant modifier, while the shares-based calculation system ensures that balance reads always reflect the

current state. The contract's internal accounting cannot be manipulated through view function calls or cross-function reentrancy attempts.

7. **Efficient Bit Operations Implementation:** The protocol implements bit operations in a secure and gas-efficient manner, particularly in the approval mechanism. In the LGEStaking contract, the use of `~uint256(0)` for maximum approval value is implemented correctly, representing the maximum possible uint256 value (`2^256 - 1`). This bitwise NOT operation on zero creates a value with all bits set to 1, which is more gas-efficient than using `type(uint256).max`. The implementation ensures safe token approvals while maintaining optimal gas consumption and preventing potential overflow scenarios in token allowances.

8. **Comprehensive Bridge Integration Security:** The protocol implements secure integrations with three distinct bridge types, each handling specific token transfers: -- The `USDXBridge` for stablecoin-to-USDX conversions with proper decimal normalization; -- The `L1LidoTokensBridge` specifically optimized for wstETH transfers with Lido protocol; -- The standard `L1StandardBridge` for general `ERC20` token bridging;

   Each bridge integration includes proper approval mechanisms using safeApprove, gas limit configurations, and L1-to-L2 address mapping validation. The system employs a unified interface pattern while maintaining specialized handling for different token types (USDC, wstETH, and standard ERC20s), ensuring secure cross-chain transfers with proper validation checks.

9. **Rebasing tokens** (based on shares) should not be added to the allowlist, since this condition may be violated.

   stETH should not be supported by the system, because: if stETH and wstETH are added to the allowlist, after depositing stETH via `depositERC20()`, the tokens can be stolen by anyone through `depositETH()` with a minimal amount followed by a `withdraw()`.

10. **Critical Decimal Precision Requirement:** The protocol's bridge implementation requires USDX to be deployed with 18 decimals on Mainnet to ensure proper value preservation during bridging operations. The current decimal normalization mechanism could lead to significant value loss if USDX is deployed with fewer decimals. For instance, if USDX uses 6 decimals while bridging from a stablecoin with 18 decimals, users would lose precision in the conversion process.

11. **Opaque Stablecoin Management in USDXBridge:** The USDXBridge contract serves as a critical component for cross-chain operations, accumulating user-deposited stablecoins while minting corresponding USDX on L2. However, the protocol lacks documentation on how the accumulated stablecoins serve as backing for minted USDX.

12. **Potential Risk of the Unlimited Minting of USDX:** It is possible for protocol owners to use the USDX minting in a malicious way, so that it is important to have the owner address protected with a multisig or DAO. This is related to the `withdrawERC20` function in the `USDXBridge` contract, which allows protocol owners to withdraw stablecoins which were used for bridging (and minting USDX on the L2 side).

---

2. FINDINGS REPORT

2.1 Critical

NOT FOUND

2.2 High

NOT FOUND

2.3 Medium

1. Potential Inflation Attack Risk

Found by @DmitriZaharov, @timofey015, @plmorozov

Merged by @plmorozov

Verified by @DmitriZaharov

Status

NEW

Description

The issue was identified within the `initialize` function in the `OzUSD` contract. Currently, this function allows for the possibility of the `_sharesAmount` variable to be set to zero during initialization. This, in turn, paves the way for a potential inflation attack, which can disrupt the economic stability of the protocol.

Even without an inflation attack, inflation itself (causing the share price to increase significantly) can be dangerous to the protocol. `wozUSD` is expected to have a price that steadily increases relative to USD, but it should not differ by several orders of magnitude.

For example, suppose the `OzUSD` contract is initialized with `_sharesAmount == 1000`. If an attacker sends 1 USDX directly to the contract (i.e., `10 ** 18` wei USDX), the price of 1 wei of `wozUSD` would become approximately equal to `10 ** 15` wei USDX. Since the `decimals` for USDX and wozUSD are the same, wozUSD would become 10 ** 15 times more expensive than USDX.

This would likely cause issues with displaying `wozUSD` balances in front-end interfaces. For example, 1000 USD would be approximately equal to `10 ** 6` wei `wozUSD`. Considering `decimals == 18`, many wallets would display this as zero.

In the example above, `OzUSD` would also face issues. Since all calculations occur at the share level, the effective precision of `OzUSD` would drop to three decimal places.

This issue is classified as **Medium** severity because, if exploited, it could severely complicate and degrade user interactions with the protocol.

Recommendation

We recommend modifying the `ozUSD` contract to make artificial inflation of the share price impossible:

1. Add an additional check in `OzUSD.initialize` to ensure that `_sharesAmount` cannot be zero.
2. Make the `distributeYield()` function permissioned by creating a dedicated role for this purpose. Store the USDX received through `distributeYield()` in storage, and use this stored value in the `_getTotalPooledUSDX()` function instead of relying on the contract's balance.
3. Remove the `receive()` function to reduce the probability of yield being mistakenly sent to the contract without being accounted for.
4. Prohibit the invocation of `mintOzUSD()` and `distributeYield()` before the contract has been initialized.

Client's Commentary

comment here

2. USDC Depeg Risk in USDX Bridge

Found by @DmitriZaharov, @timofey015

Merged by @plmorozov

Verified by @DmitriZaharov

Status

NEW

Description

The following issue has been identified within the `bridge` function of the `USDXBridge` contract. Specifically, this involves a potential instant arbitrage risk in cases where USDC or other assets lose their peg. In such cases, the market price of USDX may depreciate because the bridge will provide an instantaneous arbitrage opportunity. This represents a risk that can impact the overall stability of the USDX token.

Recommendation

We recommend implementing a `pause` function capability in the bridge functionality, allowing the operation to be temporarily halted. This could assist in risk mitigation by providing control over potential arbitrage exploitation and reducing potential market depreciation impacts.

Client's Commentary

> comment here

3. Incorrect Burn Event Emission in `redeemOzUSD` Function

Found by @timofey015

Merged by @plmorozov

Verified by @DmitriZaharov

Status

NEW

Description

This issue has been identified within the `redeemOzUSD` function of the `OzUSD` contract.

The function emits a transfer event that suggests the funds are being burned from `msg.sender`, while in reality, the funds are being burned from `_from`. This discrepancy could lead to confusion and potentially incorrect tracking of events.

The issue is classified as **Medium** severity because it impacts the reliability and accuracy of event logs, which are essential for external systems and off-chain monitoring.

Recommendation

We recommend revising the `_emitTransferEvents` function call to correctly reflect that the funds are being burned from `_from` rather than `msg.sender`.

Client's Commentary

comment here

2.4 Low

1. Absence of Pause Functionality During Minting Process

Found by @DmitriZaharov

Merged by @plmorozov

Verified by @DmitriZaharov

Status

NEW

Description

During the audit, a concern was identified in the `mintOzUSD` function of the `ozUSD` contract which lacks a pause functionality. This shortcoming becomes a concern in emergency situations where it may become critical to freeze the process of increasing the total supply of ozUSD tokens.

Recommendation

It is recommended to introduce a pause mechanism for the `mintOzUSD` function.

Client's Commentary

comment here

2. Zero Shares Transfer

Found by @DmitriZaharov

Merged by @pImorozov

Verified by @DmitriZaharov

Status

NEW

Description

A minor issue has been identified within the `_transferShares` function of the `ozUSD` contract. The function does not restrict the transfer of zero shares. This becomes problematic as it triggers non-zero token transfer events and permits free non-zero ozUSD transfers, which could be a potential loophole exploited in the system.

Recommendation

We recommend implementing a restraint in the `_transferShares` function that disallows the transfer of zero shares.

Client's Commentary

> comment here

3. Potential Balance Theft due to Zero Share Burning

Found by @DmitriZaharov

Merged by @pImorozov

Verified by @DmitriZaharov

Status

NEW

Description

An issue of Low severity has been identified in the `_burnShares` function within the `ozUSD` contract. Currently, it allows the burning of zero shares, which can be used for stealing some weis of USDX from the contract.

Recommendation

It is recommended to disallow the burning of zero shares within the `_burnShares` function.

Client's Commentary

comment here

4. Absence of a USDX to Stablecoin Exchange Mechanism

Found by @DmitriZaharov

Merged by @plmorozov

Verified by @DmitriZaharov

Status

NEW

Description

In the `USDXBridge` contract, it has been identified that there's no explicit mechanism for users to exchange their `USDX` tokens back to a deposited stablecoin. This constraint poses potential limitations to users, particularly those who do not want to complete the KYC process, which is the currently mandated pathway to perform such an exchange in case of a lack of liquidity of USDX on the market.

This issue is classified as Low severity because while it doesn’t pose a direct security risk, it can potentially impact the user experience and limit the operability of the contract by not providing an accessible pathway for USDX to stablecoin conversion.

Recommendation

We recommend adding this information to the documentation that the users might need to complete the KYC process to exchange USDX tokens.

Client's Commentary

comment here

5. Lack of Fee-on-Transfer Tokens Check

Found by @DmitriZaharov, @timofey015

Merged by @plmorozov

Verified by @DmitriZaharov

Status

NEW

Description

This issue has been identified within the `bridge` function of the `USDXBridge` contract.

The function does not account for tokens with a fee-on-transfer mechanism. For example, while the USDT contract currently has the fee-on-transfer feature disabled, if this feature were to be activated with a fee greater than 0, the `bridge` function would incorrectly mint USDX tokens at a 1:1 ratio without considering the deducted fee. This discrepancy could result in over-minting of USDX tokens and misaligned token balances.

The issue is classified as **Low** severity because the current functionality remains unaffected unless fee-on-transfer tokens are used. However, it poses a potential risk if the feature is activated in future token implementations.

Recommendation

We recommend implementing additional checks to verify the actual amount of tokens received after the transfer.

Client's Commentary

comment here

6. Inefficient Implementation of the stETH Integration

Found by @DmitriZaharov

Merged by @plmorozov

Verified by @DmitriZaharov

Status

NEW

Description

The issue was identified in the `depositETH()` function of the `LGEStaking` contract. Currently, it stakes ETH to stETH via the `stETH` contract and then wraps it to the `wstETH` token. This is not very efficient because the `wstETH` contract allows depositing ETH directly to it, which optimizes balance checks.

Recommendation

We advise refactoring the `depositETH()` function. It could directly interact with the `wstETH` contract to reduce complexity.

Client's Commentary

comment here

7. `_l2Destination` in the `migrate` Function

Found by @DmitriZaharov

Merged by @plmorozov

Verified by @DmitriZaharov

Status

NEW

Description

A potential refinement opportunity has been found in the `migrate` function of the `LGEStaking` contract. Current function parameters do not include a check to prevent certain addresses (`ozUSD` and `wozUSD` on L2) from being used as the Layer 2 (`_l2Destination`) destination for tokens migration. Although this loophole is not exploitable in the current environment, introducing the preventive measure ensures future system integrity.

Recommendation

We advise introducing a measure within the migrate function that verifies the `_l2Destination` parameter to ensure it neither matches the `ozUSD` nor the `wozUSD` addresses.

Client's Commentary

comment here

8. Unused Import

Found by @plmorozov

Merged by @plmorozov

Verified by @DmitriZaharov

Status

NEW

Description

The issue is identified in the contract `LGEStaking`. The `ISemver` interface is imported but never utilized.

Recommendation

We recommend removing the unused `ISemver` import to enhance code clarity and maintainability.

Client's commentary

comment here

9. Typographical Errors in Comments

Found by @plmorozov

Merged by @plmorozov

Verified by @DmitriZaharov

Status

NEW

Description

The issue is identified in the contract [`LGEStaking`](). The docstring comment contains a typographical error ("deposted" should be "deposited"). There is also another incorrect comment in the contract `LGEMigrationV1` at [the following line](). The docstring comment contains a typographical error ("Statked" should be "Staked").

Recommendation

We recommend correcting the typos in the comments.

Client's commentary

comment here

10. Missing Zero Address Checks in Constructor

Found by @plmorozov

Merged by @plmorozov

Verified by @DmitriZaharov

Status

NEW

Description

The issue is identified within the constructor of contract `LGEMigrationV1`. Multiple parameters, including `_owner`, `_l1StandardBridge`, `_l1LidoTokensBridge`, `_usdxBridge`, `_lgeStaking`, `_usdc`, and `_wstETH`, are not validated to ensure they are non-zero addresses. This can lead to misconfiguration and unexpected behavior if these parameters are incorrectly initialized.

The issue is classified as **Low** severity because it does not directly compromise security but can result in operational inefficiencies or require redeployment if discovered later.

Recommendation

We recommend adding `require` statements in the constructor to ensure that all addresses passed as parameters are non-zero, preventing misconfiguration and maintaining the integrity of the deployment.

Client's commentary

> comment here

11. Inefficient Inequality Operator in Deposit Check

Found by @plmorozov

Merged by @plmorozov

Verified by @DmitriZaharov

Status

NEW

Description

The issue is identified within the function `depositERC20` of contract `LGEStaking`. Currently, the deposit requirement uses a strict `<` comparison when validating the deposit against the cap, whereas `<=` should be used to correctly enforce the maximum deposit limit. The same issue is present in multiple places: `depositETH` function, `bridge` function.

This is classified as **Low** severity, because while it does not entirely break contract logic, it may not allow to fully use-up all the deposit cap.

Recommendation

We recommend changing the comparison from `<` to `<=` to ensure that deposits can reach the defined cap.

Client's commentary

comment here

12. Missing Parameters Checks in `LGEStaking` Constructor

Found by @timofey015

Merged by @plmorozov

Verified by @DmitriZaharov

Status

NEW

Description

This issue has been identified within the constructor of the `LGEStaking` contract.

The constructor lacks some validation checks for the input parameters. Specifically:

1. There is no check to ensure that `_owner` and `_wstETH` are non-zero addresses.
2. The `_tokens` array is not checked for duplicate addresses.

The issue is classified as **Low** severity because these omissions can be mitigated during deployment with careful input validation.

Recommendation

We recommend adding the following checks to the constructor:

1. Ensure `_owner` and `_wstETH` are non-zero addresses.
2. Verify that the `_tokens` array does not contain duplicate entries by adding here the following line: `require(!allowlisted[_tokens[i]], "USDXBridge: Duplicate tokens.");`

Client's Commentary

comment here

13. Improper Use of `assert` Instead of `require` in `OzUSD.redeemOzUSD()`

Found by @timofey015

Merged by @plmorozov

Verified by @DmitriZaharov

Status

NEW

Description

This issue has been identified within the `redeemOzUSD` function of the `OzUSD` contract.

The function uses the `assert` statement to check the success of the `_from.call` operation. However, `assert` is intended for checking invariants and internal consistency, and its failure consumes all remaining gas. For runtime conditions like this, `require` should be used instead to handle the failure more gracefully and provide a descriptive error message.

The issue is classified as **Low** severity because it does not impact the core functionality of the contract but can lead to poor debugging experience and unnecessary gas wastage in case of failure.

Recommendation

We recommend replacing the `assert` statement with a `require` statement.

Client's Commentary

comment here

14. Unexpected Token Transfer Recipient in `redeemOzUSD`

Found by @timofey015

Merged by @plmorozov

Verified by @DmitriZaharov

Status

NEW

Description

This issue has been identified within the `redeemOzUSD` function of the `OzUSD` contract.

The function transfers tokens to the `_from` address rather than the `msg.sender`. This behavior might not align with user expectations since typically, the caller `msg.sender` would expect to receive the redeemed tokens. This inconsistency could result in confusion or unexpected outcomes for users relying on this functionality.

The issue is classified as **Low** severity because it does not directly lead to a security vulnerability or fund loss but impacts the usability and clarity of the function's intended purpose.

Recommendation

We recommend reviewing the intended behavior of the `redeemOzUSD` function.

Client's Commentary

comment here

---

3. ABOUT MIXBYTES

MixBytes is a team of blockchain developers, auditors and analysts keen on decentralized systems. We build opensource solutions, smart contracts and blockchain protocols, perform security audits, work on benchmarking and software testing solutions, do research and tech consultancy.