decurity

Smart Contract Security Audit Report

Ozean

1. Contents

2. General Information

This report contains information about the results of the security audit of the Ozean (hereafter referred to as "Customer") L2 smart contracts, conducted by Decurity in the period from November 24 to November 29, 2024. The additional re-evaluation done on February 17, 2025, after a deprecation update.

2.1. Introduction

Tasks solved during the work are:

- Review the protocol design and the usage of 3rd party dependencies,
- Audit the contracts implementation,
- Develop the recommendations and suggestions to improve the security of the contracts.

2.2. Scope of Work

The audit scope included the contracts in the following repository: https://github.com/LayerLabs-app/Ozean-Contracts. Initial review was done for the commit 3d76b8077d778937eb76f83fe7b88d49e872a396.

After the audit was finished, part of the scope was deprecated due to the changes in the business requirements and because the custom gas tokens were deprecated. The re-test was done for the commit 7efbe42f70dadb9db4f00a226e575349d5d35cff.

2.3. Threat Model

The assessment presumes the actions of an intruder who might have the capabilities of any role (an external user, token owner, token service owner, or a contract). The risks of centralization were not taken into account at the Customer's request.

The main possible threat actors are:

- User,
- Protocol owner,

- Liquidity Token owner/contract,
- Bridge.

The table below contains sample attacks that malicious attackers might carry out.

Table. Theoretically possible attacks

Attack	**Actor**
Contract code or data hijacking *Deploying a malicious contract or submitting malicious data*	Contract owner Token owner
Financial fraud *A malicious manipulation of the business logic and balances, such as a reentrancy attack or a flash loan attack*	Anyone
Attacks on implementation *Exploiting the weaknesses in the compiler or the runtime of the smart contracts*	Anyone

2.4. Weakness Scoring

An expert evaluation scores the findings in this report, and the impact of each vulnerability is calculated based on its ease of exploitation (based on the industry practice and our experience) and severity (for the considered threats).

2.5. Disclaimer

Due to the intrinsic nature of the software and vulnerabilities and the changing threat landscape, it cannot be generally guaranteed that a certain security property of a program holds.

Therefore, this report is provided “as is” and is not a guarantee that the analyzed system does not contain any other security weaknesses or vulnerabilities. Furthermore, this report is not an endorsement of the Customer’s project, nor is it an investment advice.

That being said, Decurity exercises the best effort to perform its contractual obligations and follow the industry methodologies to discover as many weaknesses as possible and maximize the audit coverage using limited resources.

3. Summary

As a result of this work, we haven’t discovered exploitable critical exploitable security vulnerabilities but discovered a few low and medium severity issues.

The other suggestions included fixing the low-risk issues and some best practices (see Security Process Improvement).

After the deprecation of the custom gas tokens, part of the previously audited code became obsolete. Additional notes as of February 18, 2025:

1) IstETH and IwstETH interfaces are redundant nowm
2) If the LGEStaking contract was ever deployed and used, the previous eth deposits should be handled correctly further (they are already accounted for as wstETH token deposits),
3) If the contract was ever deployed and used, there's extra non-needed stETH approve.

3.1. Suggestions

The table below contains the discovered issues, their risk level, and their status as of December 02, 2024.

Table. Discovered weaknesses

Issue	Contract	Risk Level	Status
Tokens can become irretrievable due to incorrect l1-l2 address mapping or bridge failures	src/L1/LGEMigrationV1.sol	**Medium**	Fixed
Centralization risks	src/L1/LGEStaking.sol src/L1/USDXBridge.sol	**Low**	Acknowledged
stETH can be sweeped from LGEStaking	src/L1/LGEStaking.sol	**Low**	Acknowledged
Lack of restriction for fee-on-transfer tokens deposit	src/L1/LGEStaking.sol	**Low**	Acknowledged

Issue	Contract	Risk Level	Status
Blacklisted tokens may lock user funds	src/L1/LGEStaking.sol	Info	Acknowledged
Potential precision loss	src/L2/OzUSD.sol	Info	Acknowledged
No msg.value check in `distributeYield`	src/L2/OzUSD.sol	Info	Fixed
Hardcoded gas limit	src/L1/LGEMigrationV1.sol src/L1/USDXBridge.sol	Info	Fixed
Inconvenient to withdraw exact amount in shares	src/L2/OzUSD.sol	Info	Acknowledged
SafeERC20 not used for approve in Migration	src/L1/LGEMigration.sol	Info	Fixed
Unnecessary approve is required to redeem ozUSD	src/L2/OzUSD.sol	Info	Fixed

4. General Recommendations

This section contains general recommendations on how to improve the overall security level.

The Findings section contains technical recommendations for each discovered issue.

4.1. Security Process Improvement

The following is a brief long-term action plan to mitigate further weaknesses and bring the product security to a higher level:

- Keep the whitepaper and documentation updated to make it consistent with the implementation and the intended use cases of the system,
- Perform regular audits for all the new contracts and updates,
- Ensure the secure off-chain storage and processing of the credentials (e.g. the privileged private keys),
- Launch a public bug bounty campaign for the contracts.

5. Findings

5.1. Tokens can become irretrievable due to incorrect l1-l2 address mapping or bridge failures

Risk Level: **Medium**

Status: Fixed

Contracts:

- src/L1/LGEMigrationV1.sol

Description:

During the deployment of the migration contract, the admin manually maps L1 token addresses to L2 token addresses. If the admin makes an error in this mapping, the bridge will fail during token migration. In such a scenario, the tokens are returned to the sender (the migration contract). However, since the migration contract lacks an `extract` or equivalent function to recover tokens, they will remain permanently locked in the contract.

Additionally, this issue applies to any bridge transaction failures. If the transaction reverts for any reason (e.g., insufficient gas limit, mismatched token pair, or bridge-specific issues), the tokens will return to the migration contract and become irretrievable.

Impact:

- Tokens can be permanently locked in the migration contract if:
  - Incorrect L1-L2 address mapping is set during deployment.
  - Bridge transactions fail for any reason.
- Users lose access to their funds, and the protocol cannot recover the locked tokens.

Remediation:

1. **Validation During Deployment**: Add a validation mechanism to ensure the correctness of L1-L2 address mapping at deployment.
   - Example: Use an off-chain tool or automated tests to verify address mappings.

2. **Implement Token Recovery**: Add a function to the migration contract that allows an admin or trusted role to recover tokens stuck in the contract due to failed transactions.

```
function extractTokens(address _token, uint256 _amount, address _recipient)
external onlyAdmin {
    IERC20(_token).transfer(_recipient, _amount);
}
```

1. **Enhanced Bridge Error Handling**: Log detailed events for failed transactions, including the reason for failure, to allow for faster resolution.

By implementing these changes, the protocol can prevent or mitigate token lock scenarios and maintain user trust.

5.2. Centralization risks

Risk Level: Low

Status: Acknowledged

Contracts:

- src/L1/LGEStaking.sol
- src/L1/USDXBridge.sol

Description:

Having a single EOA as the only owner of contracts is a large centralization risk and a single point of failure. A single private key may be taken in a hack, or the sole holder of the key may become unable to retrieve the key when necessary. Consider changing to a multi-signature setup, or having a role-based authorization model.

Here're some of the critical functions callable by the owner:

```
File: src/L1/LGEStaking.sol
175:     function setAllowlist(address _token, bool _set) external onlyOwner {
183:     function setDepositCap(address _token, uint256 _newDepositCap)
external onlyOwner {
193:     function setMigrationContract(address _contract) external onlyOwner {
200:     function setPaused(bool _set) external onlyOwner {
```

175, 183, 193, 200.

```
File: src/L1/USDXBridge.sol
136:     function setAllowlist(address _stablecoin, bool _set) external
onlyOwner {
144:     function setDepositCap(address _stablecoin, uint256 _newDepositCap)
external onlyOwner {
152:     function withdrawERC20(address _coin, uint256 _amount) external
onlyOwner {
```

136, 144, 152.

Remediation:

To mitigate these risks, consider implementing one or more of the following measures:

- **Multi-Signature Wallet**: Replace the single EOA owner with a multi-signature wallet that requires multiple parties to authorize critical actions.
- **Role-Based Access Control (RBAC)**: Introduce a more granular role-based authorization model, allowing different roles (e.g., admin, manager) to perform specific actions without centralizing all permissions under one account.
- **Timelocks**: Add time-delayed execution for sensitive functions to reduce the risk of immediate exploitation if the owner key is compromised.

5.3. stETH can be sweeped from LGEStaking

Risk Level: Low

Status: Acknowledged

Contracts:

- src/L1/LGEStaking.sol

Description:

The `depositETH()` function wraps the entire `stETH` balance of the contract into `wstETH`, not just the newly deposited amount. This could potentially allow an attacker to sweep any manually deposited `stETH` from the contract by calling `depositETH()` with a minimal amount of ETH.

```
function depositETH() external payable nonReentrant whenNotPaused {
        require(!migrationActivated(), "LGE Staking: May not deposit once
migration has been activated.");
        require(msg.value > 0, "LGE Staking: May not deposit nothing.");
        require(allowlisted[wstETH], "LGE Staking: Token must be
allowlisted.");
        IstETH(stETH).submit{value: msg.value}(address(0));
        uint256 wstETHAmount =
IwstETH(wstETH).wrap(IstETH(stETH).balanceOf(address(this))); // @audit all
balance of stETH is converted to wstETH
        require(
            totalDeposited[wstETH] + wstETHAmount < depositCap[wstETH],
            "LGE Staking: deposit amount exceeds deposit cap."
        );
        balance[wstETH][msg.sender] += wstETHAmount;
        totalDeposited[wstETH] += wstETHAmount;
        emit Deposit(wstETH, wstETHAmount, msg.sender);
    }
```

Remediation:

Consider wrapping the newly deposited amount.

```
function depositETH() external payable nonReentrant whenNotPaused {
    // ... existing checks ...
    uint256 preBalance = IstETH(stETH).balanceOf(address(this));
    IstETH(stETH).submit{value: msg.value}(address(0));
    uint256 postBalance = IstETH(stETH).balanceOf(address(this));
    uint256 wstETHAmount = IwstETH(wstETH).wrap(postBalance - preBalance);
    // ... rest of the function ...
}
```

5.4. Lack of restriction for fee-on-transfer tokens deposit

Risk Level: Low

Status: Acknowledged

Contracts:

- src/L1/LGEStaking.sol

Description:

As stated in the optimism standard bridge documentation, fee-on-transfer tokens are not supported. However, the contract allows them to be deposited if they are allowed by the owner.

As a result:

- Deposits and withdrawals may result in incorrect balances for fee-on-transfer tokens.
- Latent funds left in the contract due to deducted fees may cause subsequent operations to succeed unexpectedly.

Remediation:

Consider restricting fee-on-transfer tokens deposit.

```
function depositERC20(address _token, uint256 _amount) external nonReentrant
whenNotPaused {
    // ... existing checks ...
    require(
        totalDeposited[_token] + _amount < depositCap[_token],
        "LGE Staking: deposit amount exceeds deposit cap."
    );
    uint256 balanceBefore = IERC20(_token).balanceOf(address(this));
    IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
    require(IERC20(_token).balanceOf(address(this)) - balanceBefore ==
_amount, "LGE Staking: Fee-on-transfer tokens not supported")

    balance[_token][msg.sender] += _amount;
    totalDeposited[_token] += _amount;
    emit Deposit(_token, _amount, msg.sender);
}
```

References:

- https://docs.optimism.io/builders/app-developers/bridging/standard-bridge

5.5. Blacklisted tokens may lock user funds

Risk Level: Info

Status: Acknowledged

Contracts:

- src/L1/LGEStaking.sol

Description:

The `withdraw` function relies on the `IERC20.safeTransfer` method to transfer tokens back to users. However, if the token implements a blacklist functionality, such as blocking transfers for certain addresses, users who are blacklisted may be unable to withdraw their funds. This could result in permanently locked user balances for such tokens.

Impact:

- **User Funds Locked**: Users who are blacklisted by specific token contracts will be unable to withdraw their funds, effectively trapping their assets in the contract.

- **Protocol Trust Risk**: Users may lose confidence in the protocol if they cannot access their deposited funds due to external token mechanics.

Remediation:

Consider allowing the user to specify the recipient of the funds.

```
function withdraw(address _token, address recipient, uint256 _amount) external
nonReentrant whenNotPaused {
    require(_amount > 0, "LGE Staking: may not withdraw nothing.");
    require(balance[_token][msg.sender] >= _amount, "LGE Staking: insufficient
deposited balance.");

    balance[_token][msg.sender] -= _amount;
    totalDeposited[_token] -= _amount;
    IERC20(_token).safeTransfer(recipient, _amount)
    emit Withdraw(_token, sentAmount, recipient);
}
```

5.6. Potential precision loss

Risk Level: Info

Status: Acknowledged

Contracts:

- src/L2/OzUSD.sol

Description:

The smart contract's calculation of shares from assets (and vice versa) does not implement proper rounding rules, as specified in OpenZeppelin's ERC4626 standard. This lack of rounding may lead to edge cases where small discrepancies are exploited to gain virtual shares or assets.

Impact:

- **Exploitation Risk**: Users may exploit rounding inconsistencies to mint more shares than entitled or withdraw more assets than they deposited.
- **Loss of Funds**: Over time, repeated exploitation can lead to financial discrepancies in the protocol.

Remediation:

Adopt consistent rounding rules in share and asset calculations to avoid discrepancies. Follow the implementation provided in OpenZeppelin's ERC4626 to prevent rounding errors.

Reference: [OpenZeppelin ERC4626 Rounding Rules]()

5.7. No msg.value check in `distributeYield`

Risk Level: Info

Status: Fixed

Contracts:

- src/L2/OzUSD.sol

Description:

The `distributeYield` function serves no purpose other than emitting an event. Since it can be called by anyone and accepts arbitrary ETH, it is vulnerable to event spamming attacks. Malicious actors can repeatedly call this function with 0 amounts of ETH, causing unnecessary on-chain event logs.

Remediation:

Check that `msg.value` is greater than 0.

5.8. Hardcoded gas limit

Risk Level: Info

Status: Fixed

Contracts:

- src/L1/LGEMigrationV1.sol
- src/L1/USDXBridge.sol

Description:

Different tokens and bridges may have varying gas requirements, and hardcoding limits such as `21000` can result in failed transactions if the actual gas needed exceeds the value.

```
constructor(
        address _owner,
        address _l1StandardBridge,
        address _l1LidoTokensBridge,
        address _usdxBridge,
        address _lgeStaking,
        address _usdc,
        address _wstETH,
        address[] memory _l1Addresses,
        address[] memory _l2Addresses
    ) {
        ...
        for (uint256 i; i < length; ++i) {
            l1ToL2Addresses[_l1Addresses[i]] = _l2Addresses[i];
            gasLimits[_l1Addresses[i]] = 21000;
        }
    }
```

For the Lido bridge, a formula is used to calculate the recommended minimum gas, which significantly exceeds the set limit of 21,000: `(L2Bridge.finalizeDeposit() + OptimismPortal.minimumGasLimit(depositData.length)) * 1.5`,

However, in the current implementation, only 21,000 gas is allocated, equivalent to `OptimismPortal.minimumGasLimit(0)`. This amount may be insufficient for the transaction to be completed on the destination chain.

```
l1LidoTokensBridge.depositERC20To(
    _tokens[i], l1ToL2Addresses[_tokens[i]], _l2Destination, _amounts[i],
gasLimits[_tokens[i]], ""
);
```

Impact:

- Transactions may fail if the hardcoded gas limit is insufficient for execution, particularly for complex token transfers or interactions with bridges that require higher gas.
- The system becomes less flexible and more prone to future failures as gas requirements evolve or vary for different tokens.

Remediation:

Gas limits can vary depending on the token, as additional operations may be required for contracts on the destination chain. It is advisable to either increase the gas limit uniformly for all tokens or set the gas limit for `wstETH` to the recommended value.

```
constructor(
        ...
        for (uint256 i; i < length; ++i) {
            l1ToL2Addresses[_l1Addresses[i]] = _l2Addresses[i];
-           gasLimits[_l1Addresses[i]] = 21000;
+           gasLimits[_l1Addresses[i]] = 300000;
        }
    }
```

Or set recommended gas limit for `wstETH` upon construction:

```
constructor(
        ...
        for (uint256 i; i < length; ++i) {
            l1ToL2Addresses[_l1Addresses[i]] = _l2Addresses[i];
            gasLimits[_l1Addresses[i]] = 21000;
        }
+       gasLimits[wstETH] = 300000
    }
```

5.9. Inconvenient to withdraw exact amount in shares

Risk Level: Info

Status: Acknowledged

Contracts:

- src/L2/OzUSD.sol

Description:

The current withdrawal mechanism requires users to estimate the `_ozUSDAmount` to withdraw the maximum number of shares. This guessing process can be challenging to perform accurately off-chain, leading to inefficiencies or failed transactions.

Impact:

- Users may struggle to calculate the correct `_ozUSDAmount` required for their desired withdrawal.
- Off-chain computations can result in overestimation or underestimation, complicating user experience.

Remediation:

Add a new function, similar to `redeem` in OpenZeppelin's ERC4626 implementation, allowing users to specify the exact number of shares to withdraw. This will streamline the withdrawal process and improve usability.

This enhancement ensures users can withdraw the exact amount they need without requiring complex off-chain calculations.

5.10. SafeERC20 not used for approve in Migration

Risk Level: **Info**

Status: Fixed

Contracts:

- src/L1/LGEMigration.sol

Description:

The `migrate` function uses `IERC20.approve` to grant token allowances. This approach is unsafe because it does not handle situations where the current allowance is non-zero. If the allowance is already set to a non-zero value, calling approve could fail or allow for potential race conditions, leading to unexpected behavior.

For example:

```
IERC20(_tokens[i]).approve(address(l1StandardBridge), _amounts[i]);
```

Impact:

- Some tokens (like USDT) may not be approved successfully if the current allowance is non-zero, resulting in failed operations.
- Potential for malicious actors to exploit race conditions during allowance updates.

Remediation:

Replace all instances of `approve` with `safeApprove` from the `SafeERC20` library to ensure allowances are safely set. Alternatively, use `forceApprove` patterns or first set the allowance to `0` before setting a new allowance.

Example fix:

```
IERC20(_tokens[i]).forceApprove(address(l1StandardBridge), _amounts[i]);
```

This approach ensures that token approvals are safely managed and reduces the risk of failed transactions or unintended exploits.

5.11. Unnecessary approve is required to redeem ozUSD

Risk Level: Info

Status: Fixed

Contracts:

- src/L2/OzUSD.sol

Description:

The `redeemOzUSD()` function requires an approval even when the caller is redeeming their own tokens. This is implemented through the `_spendAllowance()` check:

```
function redeemOzUSD(address _from, uint256 _ozUSDAmount) external
nonReentrant {
    // ... existing code ...
    _spendAllowance(_from, msg.sender, _ozUSDAmount);
    // ... existing code ...
}
```

This is unnecessary and creates friction for users because:

- When `msg.sender` is redeeming their own tokens (msg.sender == _from), they shouldn't need to approve themselves
- This forces users to execute two transactions (approve + redeem) instead of one when redeeming their own tokens

Remediation:

Consider adding a condition to skip the allowance check when the caller is the token owner:

```
function redeemOzUSD(address _from, uint256 _ozUSDAmount) external
nonReentrant {
    require(_ozUSDAmount != 0, "OzUSD: Amount zero.");
    if (msg.sender != _from) {
        _spendAllowance(_from, msg.sender, _ozUSDAmount);
    }
    // ... rest of the function ...
}
```

6. Appendix

6.1. About us

The Decurity team consists of experienced hackers who have been doing application security assessments and penetration testing for over a decade.

During the recent years, we've gained expertise in the blockchain field and have conducted numerous audits for both centralized and decentralized projects: exchanges, protocols, and blockchain nodes.

Our efforts have helped to protect hundreds of millions of dollars and make web3 a safer place.