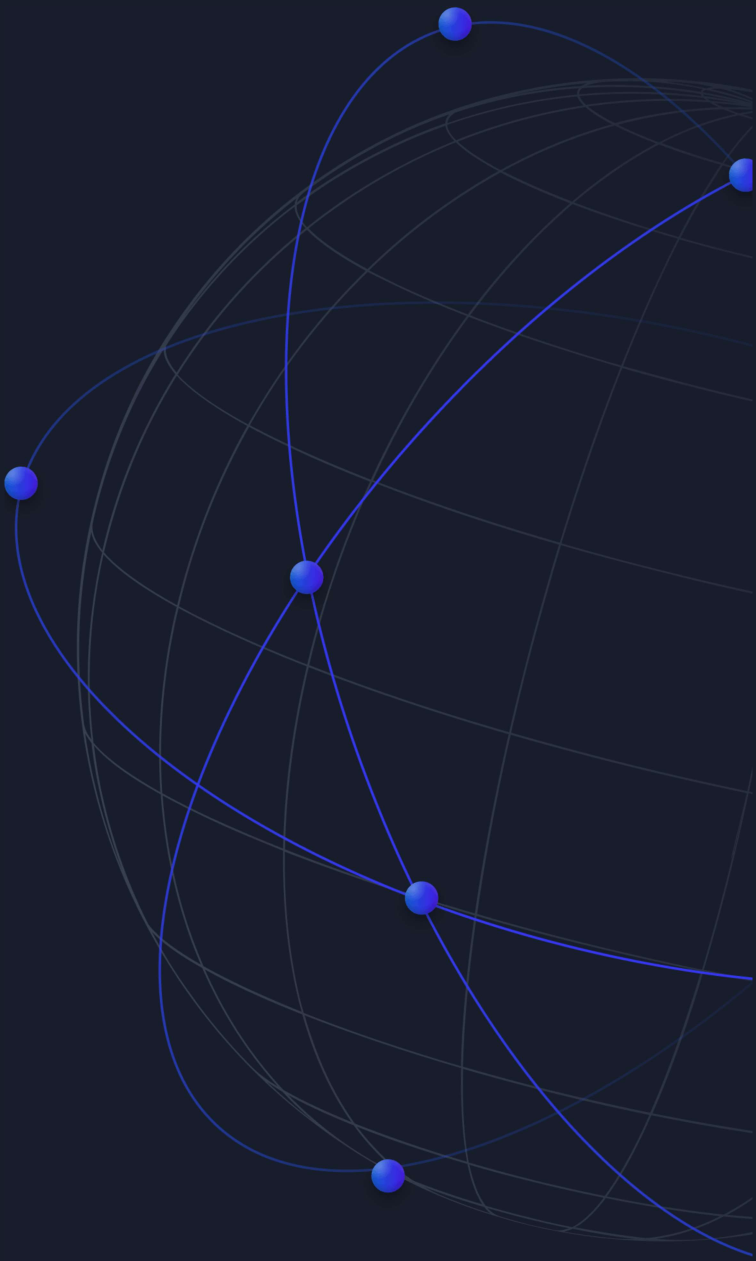
Black paper

Stake2Care

Security audit



Security audit
25/06/2024

Summary

I. Introduction

1. About Black Paper

Developing crypto projects is hard. Keeping them safe and secure is even harder. Today, hundreds of hacks happen on a regular basis in the crypto space. In 2023, more than $2B of users' funds were lost or stolen.

Black Paper was founded to empower builders to securely launch and operate their blockchain-based projects.

Cybersecurity requires specific expertise which is very different from smart contract development's logic. Our strong background in security allows us to work hand-to-hand with projects to ensure a smooth and secure development of their code. To do so, we leverage a team of smart contracts researchers who have experience in "white-hacking" competitions and bug bounty programs.

In less than 10 months since Black Paper' inception, we have:

- served more than 15 clients
- detected 300+ vulnerabilities
- protected $40M of users' funds

2. Methodology

a. Preparation

Effective audit preparation is an integral component that holds equal significance as the audit process itself. Whilst undergoing the preparation phase, customers thoroughly scrutinize their projects to ascertain that each of their planned strategies has been meticulously executed. This stage facilitates the detection of a myriad of issues by the clients themselves.

In order to derive optimum benefits from the audit, the clients should adhere to the following checklist:

- Draft a comprehensive list of functional requisites for the project. It has been done during the first preparation meeting (18/06/2024)
- Prepare a detailed technical description of the project.
- Establish an efficient development environment for the project.
- Develop and execute unit tests.
- Ensure that the code adheres to the best practices and security standards.

All was done in the really well detailed readme.md file and during the first technical meeting (18/06/2024).

b. Review

The code review process constituted a fundamental component of the audit, involving a detailed examination of the solidity codebase to ensure adherence to security best practices, mitigate potential vulnerabilities, and enhance overall code quality. Below is a comprehensive overview of the methodologies employed and considerations addressed during the code review:

1. **Security Vulnerability Assessment:**

   The primary objective of the code review is to identify and mitigate security vulnerabilities that could potentially compromise the integrity and functionality of the smart contracts. This involved a meticulous examination of the code for common vulnerabilities such as reentrancy, arithmetic overflow/underflow, improper access control, and unchecked external calls.

2. **Reentrancy Mitigation:**

   Reentrancy vulnerabilities, which allow malicious actors to manipulate contract state by repeatedly re-entering a function before its execution is complete, is thoroughly scrutinized. Critical functions such as token transfers and state modifications are assessed to ensure that appropriate checks and safeguards are in place to mitigate the risk of reentrancy attacks.

3. **Access Control Logic:**

   Access control mechanisms, particularly role-based access control, are carefully reviewed to verify that permissions are correctly assigned and enforced. Special attention are given to roles such as the admin.

4. **Input Validation and Sanitization:**

   Input validation are integral to the code review process to prevent potential exploits arising from malicious or unexpected user input. Parameters passed to functions are scrutinized to ensure that they are properly validated and sanitized to mitigate the risk of input-based vulnerabilities.

5. **Gas Optimization and Efficiency:**

   Gas optimization techniques are evaluated to ensure that the smart contracts are efficiently designed and minimize transaction costs for users. Code segments are analyzed to identify opportunities for optimization, including reducing redundant computations, minimizing storage usage, and optimizing loop iterations.

6. **Code Readability and Documentation:**

   The clarity and readability of the codebase are assessed to facilitate ease of understanding and maintenance. Clear and descriptive variable names, well-structured functions, and comprehensive comments/documentation are considered essential for enhancing code readability and ensuring that the logic and purpose of the code are easily discernible.

7. **Best Practices Adherence:**

   Adherence to industry best practices and standards, as outlined in the Ethereum Solidity Style Guide and other relevant guidelines, are verified throughout the code review process. Conformance to established best practices ensures code consistency, reduces the likelihood of errors, and promotes overall code quality.

8. **External Dependencies and Interactions:**

   Interactions with external contracts and dependencies are scrutinized to assess their security implications and potential impact on the smart contracts' integrity and functionality. Any external calls or dependencies are thoroughly vetted to ensure that they do not introduce vulnerabilities or expose the contracts to potential exploits.

By conducting a comprehensive code review encompassing these considerations and methodologies, the audit aimed to provide with actionable insights and recommendations to enhance the security and reliability of their smart contracts. If you have any further inquiries or require additional details regarding the code review process, please feel free to reach out.

c. Reporting

Every point in the code is subject to internal discussions with the team. At this stage, a majority of the probable issues have already been identified and documented.

Post the completion of the code review, analysis, and testing, we prepare a report. For each vulnerability, it contains the following informations.

- Description
- Severity
- Recommendation

Here are severity score definitions.

Critical

A critical vulnerability is a severe issue that can cause significant damage to the contract and its users. These vulnerabilities are easy to exploit and can result in the loss of funds, theft of sensitive data, or other serious consequences. Immediate attention is required to address these vulnerabilities.

Major

A major vulnerability is an issue that can cause significant problems for the contract and its users, but not to the same extent as a critical vulnerability. These vulnerabilities are also easy to exploit and may result in the loss of funds or other negative consequences, but they can be mitigated with timely action.

Medium

A medium vulnerability is an issue that could potentially cause problems for the contract and its users, but the difficulty to exploit is higher than major or critical vulnerabilities. These vulnerabilities may pose a risk to the contract's functionality or security, but they can be addressed without causing significant disruption.

Low

A low vulnerability is a minor issue that does not pose a significant risk to the contract or its users. These vulnerabilities are difficult to exploit and may be cosmetic or technical in nature, but they do not compromise the contract's security or functionality.

Informational

An informational finding is not a vulnerability but rather a suggestion or recommendation for improvement. These findings may include best practices for contract design, suggestions for improving code readability, or other non-critical issues. While not urgent, addressing these findings can help to optimize the contract's performance and reduce the risk of future vulnerabilities.

3. Disclaimer

In this audit, we sent all vulnerabilties found by our team. **We can't guarantee all vulnerabilities have been found.**

4. Scope

Defining the scope of a smart contract audit is crucial for ensuring that all aspects of the contract are thoroughly reviewed and any potential issues are identified. At Black Paper, we highlight the importance of defining the scope upfront and ensuring that it is comprehensive enough to cover all relevant components of the contract. This helps us to provide a comprehensive audit report that accurately identifies any potential vulnerabilities and provides recommendations for remediation.

The files included in the scope (commit 9a1ee20541ae0a7655008b191f835f531fc16dc3) are:

- ImpactVault.sol
- ImpactVaultDepositor.sol
- LidoImpactVaultDepositor.sol
- intf/IImpactVault.sol
- intf/IImpactVaultDepositor.sol

II. Vulnerabilities

LOW-1 Potential rounding issues on withdraw and redeem Fixed

Impact: Low

Description:

The `MIN_DEPOSIT` parameter partially resolves the issue related to rounding errors during `stETH` transfers as explained in the Lido documentation. However, users can still withdraw or redeem any amount without a minimum, potentially leading to a scenario where they deposit `x stETH` but withdraw less due to rounding issues.

Recommendation:

To mitigate this issue, two approaches can be considered:

1. **User Responsibility**: Assume that the user (or the front end for deposit) is responsible for managing this issue. This approach requires users to be aware of the potential rounding discrepancies and handle them appropriately. To make sure the user has enough information, we strongly recommend to add a warning on the function comments.

```
    /// @notice Make sure you don't withdraw a too low asset amount. Due to stETH
rounding issue, you could withdraw less than expected.
```

2. **Smart Contract Enforcement**: Enhance the smart contract to check the asset amount and the remaining assets for a user after a withdrawal. If the remaining assets are less than the `MIN_DEPOSIT` parameter, revert the transaction or force the withdrawal of all shares. This approach ensures that users do not end up with less stETH than they deposited due to rounding issues.

```
    /** @dev See {IERC4626-withdraw}. */
    /// @notice Withdraws assets to receiver address against owner vault shares
    /// @dev normally asset is positively rebasing and we obtain 1 asset per canceled
vault share, we however adjust the price if there was an adverse rebasing
    /// @param assets Amount of Assets to obtain
    /// @param receiver Address receiving the withdrawn assets
    /// @param owner Address owning the vault shares to burn
    function withdraw(
        uint256 assets,
        address receiver,
        address owner
    ) public override(ERC4626) returns (uint256) {
        uint256 shares = _collectDonationsAndConvertToShares(
            assets,
            Math.Rounding.Up
        );
        if (assets <= MIN_DEPOSIT) revert WithdrawTooLow();
        _withdraw(_msgSender(), receiver, owner, assets, shares);
        if (balanceOf(owner) <= MIN_DEPOSIT && balanceOf(owner) != 0) revert
InsufficientRemainingAssets();
        return shares;
    }

    /** @dev See {IERC4626-redeem}. */
    /// @notice Cancels amount shares of msg.sender and withdraws asset
    /// @dev normally asset is positively rebasing and we obtain 1 asset per canceled
vault share, we however adjust the price if there was an adverse rebasing
    /// @param shares Amount of Vault Shares to Cancel
    /// @param receiver Address receiving the withdrawn assets
    /// @param owner Address owning the vault shares to burn
    function redeem(
        uint256 shares,
        address receiver,
        address owner
    ) public override(ERC4626) returns (uint256) {
        uint256 assets = _collectDonationsAndConvertToAssets(
            shares,
            Math.Rounding.Down
        );
        if (assets <= MIN_DEPOSIT) revert WithdrawTooLow();
        _withdraw(_msgSender(), receiver, owner, assets, shares);
        if (balanceOf(owner) <= MIN_DEPOSIT && balanceOf(owner) != 0) revert
InsufficientRemainingAssets();
        return assets;
    }
```

Add the following custom errors for clarity:

```
error InsufficientRemainingAssets();
error WithdrawTooLow();
```

Note that the `previewRedeem` and `previewWithdraw` function should also be modified to revert in case of insufficient asset amount.

Status:

The fix has been implemented following recommendation 1.

INF-1 Gas optimization in `collectDonations` function Fixed

Impact: Informational

Description:

The current implementation of the `collectDonations` function repeatedly computes the eligibility of surplus collection based on the timestamp every time the function is called. This redundant computation incurs unnecessary gas costs. Specifically, the line:

```
unchecked{sufficientTransfer=timeLockedSurplus_.timestamp < uint64(block.timestamp - 3 days);}
```

is recalculated on each call, which can be optimized.

Recommendation:

Optimize the timestamp handling by directly setting the next eligible collection time during the update of `timeLockedSurplus`. This will avoid redundant computations and save gas.

Here is the optimized code:

```
/// @notice Collects Asset Surplus as a donation for Owner
/// @dev Does not collect if 24-hour timeLocked surplus is less than minimalTransfer
/// @dev At most collects Once every 3 days
/// @dev caller indicates minimalTransferAmount for computation to take place - if 0 is indicated
we revert to default minimum (as registered in storage)
function collectDonations(
    uint64 minimalTransfer
)
    public
    override(IImpactVault)
    returns (
        uint128 collectedAmount,
        uint256 totalAssets_,
        uint256 totalSupply_
    )
{
    totalSupply_ = totalSupply();
    totalAssets_ = totalAssets();
    TimelockedSurplus memory timeLockedSurplus_ = timeLockedSurplus;
    minimalTransfer = minimalTransfer == 0
        ? timeLockedSurplus_.minimalCollectAmount
        : minimalTransfer;
    if (totalAssets_ > totalSupply_ + minimalTransfer) {
        // Check if current surplus is high enough
        bool sufficientTransfer;
        unchecked { sufficientTransfer = timeLockedSurplus_.timestamp < uint64(block.timestamp);
}
        if (sufficientTransfer) {
            // 3 days TimeLock on surplus distribution - to avoid donor loss in case of potential
NAV up-down bounce
            uint128 newSurplus;
            unchecked { newSurplus = uint128(totalAssets_ - totalSupply_); }
            collectedAmount = newSurplus > timeLockedSurplus_.surplus
                ? timeLockedSurplus_.surplus
                : newSurplus;
            if (collectedAmount > minimalTransfer) {
                IERC20(asset()).safeTransfer(owner(), collectedAmount);
            } else {
                collectedAmount = 0;
            }
            unchecked {
                timeLockedSurplus = TimelockedSurplus(
                    newSurplus - collectedAmount,
                    uint64(block.timestamp) + 3 days,
                    timeLockedSurplus_.minimalCollectAmount
                );
            }
        } // Do nothing if timeLock not elapsed
    }
}
```

Following the tests here are the gas cost before the modification:

Contracts / Methods	Min	Max	Avg	# calls
ImpactVault				
collectDonations	-	-	37,430	1
deposit	83,998	113,422	109,744	16
mint	66,943	74,467	70,705	4
redeem	-	-	93,086	2
setAutoCollectThreshold	47,490	47,514	47,506	6
withdraw	-	-	93,152	2

And after the modification:

Contracts / Methods	Min	Max	Avg	# calls
ImpactVault				
collectDonations	-	-	37,427	1
deposit	83,998	113,422	109,744	16
mint	66,943	74,467	70,705	4
redeem	-	-	93,083	2
setAutoCollectThreshold	47,490	47,514	47,506	6
withdraw	-	-	93,149	2

Status:

The optimization has been implemented following recommendation.

INF-2 Outdated comments on timeLock duration Fixed

Impact: Informational

Description:

The comments in the code incorrectly state that the TimeLock on surplus distribution is 1 day, whereas the actual implementation has been changed to 3 days. This discrepancy can lead to confusion and misinterpretation of the code.

Recommendation:

Update the comments to accurately reflect the current 3-day TimeLock on surplus distribution.

```
// 3 days TimeLock on surplus distribution - to avoid donor loss in case of potential NAV up-down
bounce
```

And

```
struct TimelockedSurplus {
    uint128 surplus; // TimeLocked surplus - distributable at timelock expiry (3 days)
    uint64 timestamp; // Ok until 2554  - timestamp when surplus was timelocked
    uint64 minimalCollectAmount; // Minimal Amount to auto-Collect at each deposit/ withdrawal -
can be set by _owner. uint64 -> ~ 18 wad
}
```

Status:

The comment has been corrected.