Smart Contract Code Assessment

Produced for

SwissBorg

Executive Summary

This executive summary outlines the key findings and recommendations from the security assessment conducted for SwissBorg, May 2023. The assessment's aim was to assess the security of the provided smart contracts, identify potential vulnerabilities, and propose suggestions to potentially mitigate risks.

Scope of the Assessment The assessment covered two smart contracts outlined in 1.1.

Key Findings Our comprehensive analysis of the smart contracts has brought to light several significant concerns that merit attention:

- *Security Risks.* During our review, we pinpointed three potential security vulnerabilities that could potentially compromise the integrity and safety of the system.
- *Informational Findings.* Additionally, we have come across three informational aspects that, while not immediately threatening, could be enhanced to improve the contract's code quality.

Recommendations Based on these key findings, we propose the following actions:

- *Vulnerability Remediation.* Address the identified security vulnerabilities immediately. We suggest implementing necessary code modifications and retesting to ensure these risks have been successfully rectified.

Conclusion Our assessment has revealed some security vulnerabilities in SwissBorg's smart contracts that need attention. Addressing these issues promptly will enhance the platform's security, leading to safer transactions and increasing confidence.

This executive summary provides a concise synopsis of our findings. We recommend reading the full assessment report for a comprehensive understanding of the vulnerabilities identified, potential impacts, and our suggested mitigation strategies.

We look forward to supporting SwissBorg with the remediation of these vulnerabilities and conducting further assessments to maintain the integrity and security of the smart contracts.

1. Assessment Overview

In this section, we provide an overview of the assessment scope.

1.1 Scope

The assessment was performed on the following two solidity files:

1. `./vault/EarnVault.sol`
   sha512:0x404a732a7b1183841e10364291e405b96ca7a1cc7ba20fbdff6e53992c3cc2051f982292c0f9f1b53fb3a7ad6889ec4446346036f12bf44549016a257aa115fe
2. `./yield/module/AaveYieldModule.sol`
   sha512:0x064632aa40ae163274390e7aa1242b417feda9d0383b253e94b7503e65faa7c2d3c256be4ad11f88ba04e1fda5959e100002d21cb009993f8cc298be823f53f8

The solidity compiler version 0.8.13 was chosen.

Excluded from scope

Files not specified in the scope section — particularly test files, scripts, external dependencies, and configuration files — are not included in the assessment scope.

1.2 System Overview

1.2.1 `EarnVault.sol`

The `EarnVault` contract primarily works with three roles: `OWNER_ROLE`, `MANAGER_ROLE`, and `GUARDIAN_ROLE` (referred to as owner, manager, and guardian respectively hereinafter), each with different levels of authority and responsibilities. The roles are assigned during the initialization phase, which also sets up the vault's basic parameters, such as the initial token price feed, strategy, and status.

The contract manages deposits of a specified token (the `want` token) into a specific strategy defined by a yield-generating module, which can be modified by the owner when the vault is inactive. The strategy's balance is kept track of by the vault and is updated upon each deposit or withdrawal operation.

The contract's status can be "Active" or "Inactive", with operations like deposit, withdraw, and harvest only allowed during "Active" status. In emergency situations, the vault can be stopped, and funds can be withdrawn from the strategy. The guardian role is responsible for such operations. The vault can be resumed by the owner after being made inactive.

Furthermore, the contract supports upgradeability through UUPS (Universal Upgradeable Proxy Standard), which means the underlying logic of the contract can be upgraded by the owner without affecting the state or address of the deployed contract.

The contract also includes safety features, such as the rescue function to recover stuck funds and validation checks to prevent transactions with zero-address or zero-amount.

1.2.2 AaveYieldModule.sol

AaveYieldModule acts as an interface for interacting with the Aave DeFi lending protocol. The contract is an extension of a BaseModule, and uses a variety of different contracts and interfaces to interact with the Aave protocol.

One important aspect of this contract is the initialization function, which is responsible for setting up all necessary parameters. These include various contract addresses, rewards, and fees. This function also approves the transfer of tokens (termed want) to the Aave pool.

The contract includes functions for depositing tokens into Aave and withdrawing them. The withdraw function calculates the actual amount of want tokens withdrawn. A notable feature of this contract is the getMaxWithdrawableAmount function, which calculates the maximum amount that can be withdrawn from the protocol at any given time.

The harvest function is used to claim any rewards earned from the Aave protocol. The rewards are transferred to the caller in the form of want tokens.

To help users keep track of their investments, the contract offers getBalance and getLastUpdatedBalance functions. These return the current and last updated balances of want tokens on Aave, respectively.

The contract also includes a getExecutionFee function that returns the fee required for withdrawals from Aave. In this case, the fee is set to zero.

Additionally, the contract includes a number of private helper functions. These functions calculate profit, collect pool data, get available liquidity, and handle the withdrawal process.

2. Detailed Findings

In this section, we outline the potential security risks.

2.1 EarnVault.sol

No critical risk is found.

2.2 AaveYieldModule.sol

2.2.1 Infinite Approval Risk

Description

The function `initialize` presents an issue commonly referred to as "infinite approval". Specifically, the line

```
IERC20Upgradeable(_want).safeApprove(_params._pool, type(uint256).max);
```

provisionally grants an unlimited allowance to the `_params._pool` address. This opens the potential for the contract to be subjected to risks if the `_params._pool` contract is either compromised or operates with malicious intent. With the granted unlimited approval, the `_params._pool` could theoretically drain all the `_want` tokens from this contract without necessitating any additional permissions. Although `AaveYieldModule` isn't inherently designed for asset storage, the utilization of "infinite approval" can potentially be exploited in conjunction with other vulnerabilities, such as reentrancy attacks.

Recommendation

To address this vulnerability, it is advised to set precise allowances within the `deposit` function, rather than resorting to the provision of infinite approval. It is considered a best practice to set allowances equal to exactly what is required, without any surplus. This approach ensures a more secure, predictable contract interaction while limiting the potential scope of an attack.

2.2.2 Reentrancy Risk

Description

A potential reentrancy vulnerability exists in the `_lpProfit` function. Specifically, the function call `IPoolAave(pool).withdraw(want, aumDelta, address(this))` permits external interaction that could potentially exploit the not-yet-updated state variable `lastPricePerShare`. Despite `_lpProfit` being a private function, exclusively triggered by `harvest`–a function that is protected by `onlyVault`–the risk of reentrancy persists.

This is particularly concerning if the `vault` contract is either compromised or inherently vulnerable.

Recommendation

As a mitigation measure, it is recommended to implement checks-effects-interactions pattern. The state variable `lastPricePerShare` should be updated before calling external contracts to help prevent potential reentrancy attacks.

2.2.3 Missing Slippage Protection

Description

The function `_rewardsProfit` incorporates a swap operation that unfortunately lacks an essential element of slippage protection. More specifically, the instruction

```
IDex(dex).swap(rewardBalance, rewards[0], want, address(this));
```

is tasked with swapping the accumulated rewardBalance amount of `rewards[0]` tokens for want tokens. However, in this process, no constraints are imposed on the minimum amount of want tokens expected to be received from the swap. In situations of unfavorable market conditions, this could result in receiving a significantly lower quantity of want tokens than initially anticipated. This deficiency not only exposes the contract to potential front-running and sandwich attacks, resulting in considerable losses, but it could also be exploited in conjunction with reentrancy attacks.

It is understood that in general circumstances, the exchange from `aToken` to `want` has zero slippage. However, in extreme scenarios, for example, if Aave experiences issues and redemption from `aToken` to `want` is forbidden, it may be necessary to resort to alternative exchanges. Therefore, even though the current implementation might work under regular conditions, it is prudent to prepare for exceptional circumstances by embedding slippage protection.

Recommendation

In order to mitigate this risk, it is advisable to integrate slippage protection into the swap operation. A prevalent approach to accomplish this is by setting a parameter that clearly delineates the minimum acceptable quantity of want tokens to be received from the swap. If the actual amount of tokens received falls short of this threshold, the transaction should be programmed to automatically revert. Implementing this protective measure safeguards against extreme slippage events. Consequently, the `IDex(dex).swap` function might necessitate modifications or even replacement with a function that allows for the specification of a minimum output. It is important to highlight that the chosen minimum should take into account a reasonable degree of slippage in order to prevent transaction failures amid normal market volatility.

3. Informational

In this section, we show the informational findings that are less severe than the security risks.

3.1 EarnVault.sol

3.1.1 Unchecked wantPriceFeed Initialization

Description

In the initialize function, there is an issue related to the lack of validation checks for the wantPriceFeed parameter.

Recommendation

It is recommended to add checks to validate that the wantPriceFeed parameter is a non-zero address.

3.2 AaveYieldModule.sol

3.2.1 payable Inconsistency

Description

There is an inconsistency between the withdraw and _withdraw functions concerning the payable keyword and a require check for msg.value. Specifically, the withdraw function is marked as payable, while the subsequent _withdraw function, called within withdraw, includes the line

```
require(msg.value == 0, "Aave: msg.value must be zero");
```

This line mandates that the msg.value of the transaction should be zero, essentially negating the payable keyword in the withdraw function.

However, this could be a reasonable design if the developer intends to show the detailed error message when msg.value is not zero, explicitly indicating that the function does not accept Ether, despite the payable keyword.

3.2.2 Function State Mutability

Recommendation

The function state mutability of _getAvailableLiquidity can be restricted to pure.

4. Disclaimer

The report is provided solely for informational purposes and should not be considered as an endorsement, recommendation, or any form of legal, financial, or investment advice.

The report is based on the technical analysis of the code at the time of assessment and does not account for any updates, modifications, or alterations to the code that may occur after the report date. The code was assessed “as-is” and the findings represent the state of the code at the time of the assessment.

Although every reasonable effort has been made to ensure the accuracy, completeness, and fairness of the analysis and findings contained within the report, it is provided on an "as-is" basis without any warranties, representations, or guarantees of any kind, express or implied. This includes, but is not limited to, warranties of merchantability, fitness for a particular purpose, non-infringement, accuracy, or the presence or absence of errors, whether or not discoverable.

The authors, evaluators, and any associated parties disclaim all liability for any losses, damages, costs, or expenses (including legal fees) arising directly or indirectly from the use of or reliance on the report or its findings. This includes, but is not limited to, any damage or loss caused by errors, omissions, inaccuracies, or any misleading or out-of-date information.

The reader is solely responsible for any actions or decisions taken based on the information provided in this report. It is highly recommended that, where necessary, appropriate professional advice is sought before making any decisions or taking any actions relating to the smart contract code analyzed in this report.