Audit Report

Persistence Anchor Integration Ethereum Smart Contracts

v1.0

January 25, 2022

Table of Contents

License

Disclaimer

This audit has been performed by

Oak Security

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security has been engaged by Persistence Holdings Pte Ltd to perform a security audit of the Persistence Anchor Integration smart contracts.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following GitHub repository:

https://github.com/persistenceOne/pStake-Anchor-smartContracts

Commit hash: `1c9ae62dcd428fedd51d227e01b319fc597745fa`

Scope of the audit has been limited to the Solidity files in the `contracts` folder.

NOTE: The codebase relies heavily on integration with external protocols (Wormhole and SushiSwap). Neither of these external protocols has been reviewed as part of this audit.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line by line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

The submitted codebase implements the Ethereum support for pStake's Anchor integration, which consists of an ERC-20 token that represents bAtom and a generic bAsset hub that can be used to bond whitelisted assets. The hub is integrated with Wormhole and SushiSwap.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending, Acknowledged** or **Resolved**.

Note that audits are an important step to improve the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than a security audit and vice versa.

Summary of Findings

No	Description	Severity	Status
1	Lack of slippage protection in SushiSwap integration	Minor	Resolved
2	Duplicate checks may cause unnecessary gas usage	Minor	Resolved
3	Misleading function name	Informational	Resolved
4	Incorrect comments	Informational	Resolved
5	Solidity compiler version update	Informational	Acknowledged

Code Quality Criteria

Criteria	Status	Comment
Code complexity	Medium	-
Code readability and clarity	Medium-High	-
Level of documentation	Medium	-
Test coverage	Medium-High	-

Detailed Findings

1. Lack of slippage protection in SushiSwap integration

Severity: Minor

The function `getAmountOutMin` in `contracts/BAssetHub.sol` performs a simulated swap on SushiSwap to determine the number of tokens a user will receive for a given input. This is used by `liquidateRewards` to set up a swap on SushiSwap directly, meaning the user will get whatever outcome the current pool liquidity provides, regardless of any slippage due to low liquidity.

Recommendation

Consider adding slippage protections to the SushiSwap integration by placing acceptable bounds on the out token amount.

Status: Resolved

2. Duplicate checks may cause unnecessary gas usage

Severity: Minor

The function `whitelistAsset` in `contracts/BAssetHub.sol` performs a check whether the asset is already contained in the `EnumerableSet` data-structure used. However, the `add` function already performs this check, returning `false` if the element is already present. This leads to the check being performed twice, which is an unnecessary gas expenditure.

Recommendation

Consider not performing the check and making use of the return value of the `add` function instead.

Status: Resolved

3. Misleading function name

Severity: Informational

The function `blacklistAsset` in `contracts/BAssetHub.sol` removes an asset from the whitelist. However, removing an asset from a whitelist is semantically not the same as blacklisting an asset, since the latter commonly refers to explicitly prohibiting an asset, including blocking it from being added to a whitelist.

Recommendation

Consider changing the function name to `removeFromWhitelist`.

Status: Resolved

4. Incorrect comments

Severity: Informational

The comment in line `21` of `contracts/BAssetHub.sol` is incorrect since the transfer is executed in the inverse direction as stated.

Similarly, the comment in line `462` suggests a minting operation that does not occur in the code.

Recommendation

Consider correcting the comments.

Status: Resolved

5. Solidity compiler version update

Severity: Informational

The codebase uses Solidity version 0.7. However, this implies relying on a safe math library, since compiler versions lower than 0.8 did not implement automatic overflow protection. Using automatic overflow protection could significantly simplify the code. In addition, a number of important compiler bugs have recently been fixed. Any compiler version below 0.7.6 would not flag some kinds of source code attacks.

Recommendation

Consider updating to a more recent version of Solidity. We recommend version 0.8.4 or higher.

Status: Acknowledged

Team Reply: *“We have used the compiler version 0.7.6 as recommended, which has fixed the compiler bugs. Also, since the safeMath library has already been used in these contracts, and changing them now would alter the scope, we are currently sticking to solidity 0.7.6.*