APY Stable Bridge Secure Code Review

Findings and Recommendations Report Presented to:

APY Foundation Ltd.

November 23, 2022
Version: 3.1

Presented by:

Kudelski Security, Inc.
5090 North 40th Street, Suite 450
Phoenix, Arizona 85018

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

EXECUTIVE SUMMARY

Overview

APY Foundation Ltd. engaged Kudelski Security to perform a secure code assessment of the APY Stable Bridge smart contract system for the Ethereum Virtual Machine.

The assessment was conducted remotely by the Kudelski Security Team. The source code review took place from October 7th, 2022 – November 14th, 2022, and focused on the following objectives:

- Provide the customer with an assessment of their overall security posture and any risks that were discovered within the environment during the engagement.
- To provide a professional opinion on the maturity, adequacy, and efficiency of the security measures that are in place.
- To identify potential issues and include improvement recommendations based on the result of our tests.

This report summarizes the engagement, tests performed, and findings. It also contains detailed descriptions of the discovered vulnerabilities, steps the Kudelski Security Teams took to identify and validate each issue, as well as any applicable recommendations for remediation.

This is a redacted version of the report with the impact, evidence and remediation guidance of each finding removed before public release.

Key Findings

The following are the major themes and issues identified during the testing period. These, along with other items, within the findings section, should be prioritized for remediation to reduce to the risk they pose.

- A single administrator account claims all fees. This activity should be limited by ideally requiring multiple administrators to prevent collusion or single point of failure. This can be achieved by depositing fees to a multisignature wallet, which may further be on-chain for transparency.
- Numerical errors between the theory and implementation need careful attention in DeFi applications. Tests should guarantee boundedness and accuracy over the range of inputs.
- Extensive test coverage will need to be added to prevent the possibility of attackers exploiting small differences in quantities by repeatedly calling functions. Asserting invariants, such as constraints on the changes of balances could prevent these attacks.
- Rounding errors should receive careful attention in the test coverage across the range of input variables and state variables.
- Extensive numerical analysis is necessary to quantify the errors and possible divergences
- Divergences in any implemented equations should receive careful attention.
- Careful numerical analysis which demonstrate the boundedness of errors over the range of inputs should be provided.

During the test, the following positive observations were noted regarding the scope of the engagement:

- The code was well-structured and well-commented
- Client contacts were highly collaborative with the Kudelski Security smart contract auditing team.

- The client had already implemented measures which prevented some numerical errors from being exploited
- Tests were written for most functionality

Scope and Rules Of Engagement

Kudelski performed a Secure Code Review on the APY Stable Bridge smart contract system for EVM. The files in stable-bridge/projects/evm were the target in scope for the engagement. No additional systems or resources were in scope for this assessment.

The source code was supplied through a private repository:

In-scope source code, contracts, or programs	
https://github.com/allbridge-io/stable-bridge	Commit: 8d5cf33c4946b748eed51b09ff22f5b21998f2fb

In scope for math review
https://github.com/allbridge-io/stable-bridge/blob/8d5cf33c4946b748eed51b09ff22f5b21998f2fb/projects/evm/contracts/Pool.sol

Table 1: Scope

TECHNICAL ANALYSIS & FINDINGS

During the APY Stable Bridge Secure Code Review, we discovered 1 finding that had a medium severity rating, 5 findings that had a low severity rating, as well as 9 of informational severity rating.

The following chart displays the findings by severity.

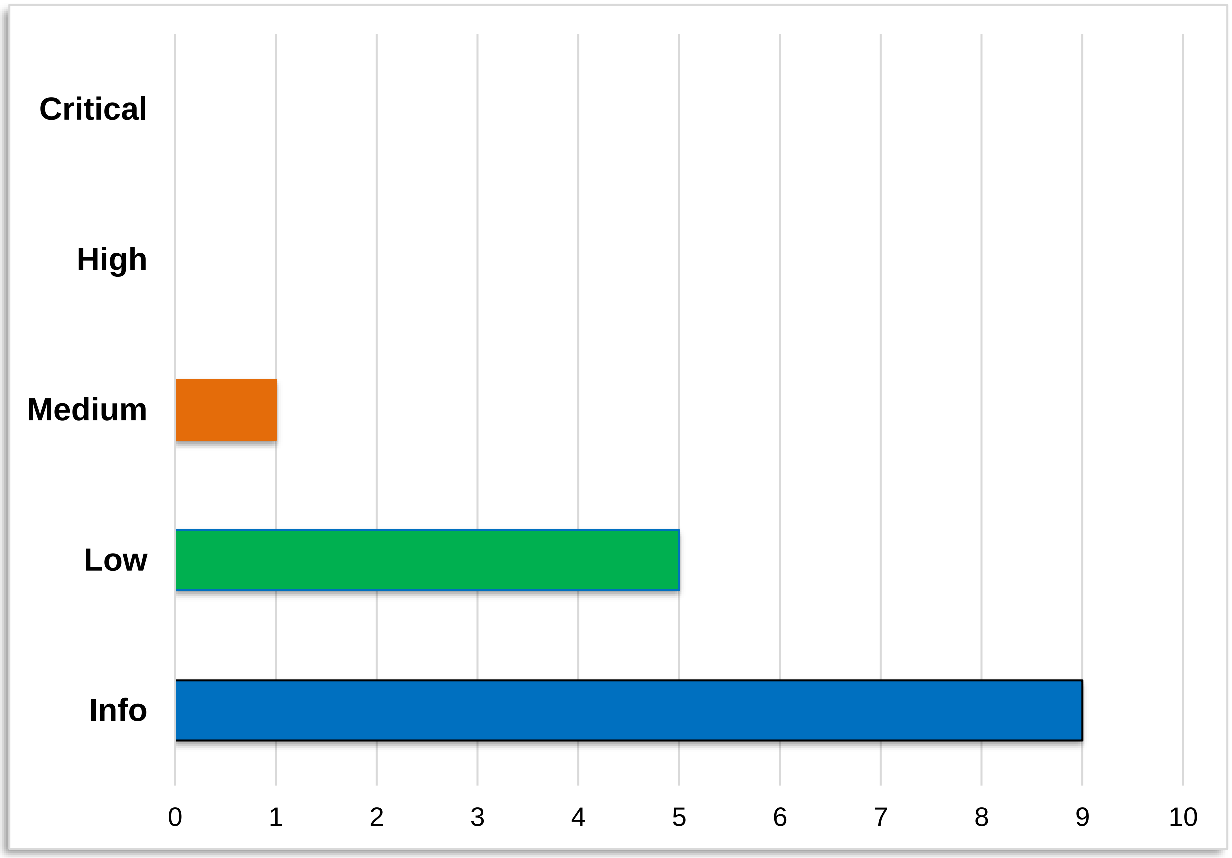

Figure 1: Findings by Severity

Findings

The *Findings* section provides detailed information on each of the findings, including methods of discovery, explanation of severity determination, recommendations, and applicable references.

The following table provides an overview of the findings.

#	Severity	Description
KS-ASB-01	Medium	Unchecked systematic errors in Pool calculations
KS-ASB-02	Low	Reentrancy from ERC20 contract during RewardManager claim admin fee
KS-ASB-03	Low	Replay across colliding chain ids
KS-ASB-04	Low	Loss of precision in Pool calculations
KS-ASB-05	Low	Outdated dependencies
KS-ASB-06	Low	Pragma solidity versions are not locked
KS-ASB-07	Informational	Library functions' first argument should be self
KS-ASB-08	Informational	No emergency stop for Pool
KS-ASB-09	Informational	No payment for publishing to Wormhole
KS-ASB-10	Informational	Shadowed constant in Pool
KS-ASB-11	Informational	Pragma ABI coder v2 is not experimental
KS-ASB-12	Informational	Pragma solidity versions are inconsistent
KS-ASB-13	Informational	Pragma solidity versions are not locked
KS-ASB-14	Informational	Reentrancy from ERC20 contract during Pool deposit
KS-ASB-15	Informational	Test suite excludes all but one test

Table 2: Findings Overview

KS-ASB-01 – Unchecked systematic errors in Pool calculations

Severity	**Medium**
Status	**Open**

Impact	**Likelihood**	**Difficulty**
High	Low	High

Description

The implementation of mathematical functions can contain numerical errors. These are differences from theoretical values produced by rounding error or other numerical approximations. This may lead to quantities such as balances which are inaccurate.

Fuctions such as *swapFromVUsd* and *swapToVUsd* alter internal balances such as *tokenBalance*. These functions also depend upon *getY*. *getY* contains rectification for rounding errors by adding one to the result, but is missing test coverage which demonstrates accurate values across all inputs. Furthermore, the calling functions *swapFromVUsd*, *swapToVUsd,* and *_preWithdrawSwap,* are missing test coverage which demonstrate the direction of rounding is appropriate for all inputs.

References

- Becoming a Millionaire, 0.000150 BTC at a Time | OtterSec (osec.io)

KS-ASB-02 – Reentrancy from ERC20 contract during RewardManager claim admin fee

Severity	**Low**
Status	**Open**

Impact	**Likelihood**	**Difficulty**
High	Low	High

Description

When calling external contracts there is a possibility that the contract will call back. If state changes are not made until the external call returns, then a reentrancy attack can occur.

The `claimAdminFee` function in the RewardManager contract call the ERC20 contract to transfer tokens from the user. After the token transfer returns, the internal state of the admin fee amount is set to zero. The ERC20 contract could use this for a reentrancy attack and claim the admin fee amount several times.

References

- Ethereum Smart Contract Best Practices: External Calls: Avoid state changes after external calls
- SWC-107: Reentrancy

KS-ASB-03 – Replay across colliding chain ids

Severity	**Low**
Status	**Open**

Impact	Likelihood	Difficulty
High	Low	Low

Description

The code use mappings of hashes to validate if a message has already been processed. The hash is calculated from token amount to transfer, recipient, source chain id, destination chain id, token contract, nonce, and messenger type.

Only the first byte from each chain id is included in the hash instead of the entire 256 bit chain id.

If bridges are deployed across chains where the first byte of the chain id collide, it will be possible to replay messages received by one of the Bridge contracts on the complementary Bridge contract.

KS-ASB-04 – Loss of precision in Pool calculations

Severity	**Low**
Status	**Open**

Impact	Likelihood	Difficulty
Low	Medium	High

Description

When using integer arithmetic it is important to do multiplication before division to avoid loss of precision due to rounding.

A simple example is the calculation $4 \cdot \frac{3}{2}$. An implementation doing multiplication before division will yield `(4 * 3) / 2 == 6`. But an implementation doing division before multiplication will yield `4 * (3 / 2) == 4` due to loss of precision.

Loss of precision occurs in the Pool contract when the field variable `d` is updated.

References

- Ethereum Smart Contract Best Practices - Integer Division

KS-ASB-05 – Outdated dependencies

Severity	**Low**
Status	**Open**

Impact	**Likelihood**	**Difficulty**
Medium	Low	High

Description

It is always a best practice to keep dependencies up-to-date to ensure all known bugs and vulnerabilities are fixed. Several dependencies can be upgraded, including some with known vulnerabilities.

References

- Ethereum Smart Contract Best Practices - Stay up to Date
- Ethereum Releases and Bugfixes
- Security Alerts | Solidity Blog (soliditylang.org)

KS-ASB-06 – Pragma solidity versions are not locked

Severity	**Low**
Status	**Open**

Impact	**Likelihood**	**Difficulty**
Low	Low	Medium

Description

It is best practice to deploy smart contracts that have been built with the exact same version of the compiler that have been used for testing. Furthermore, vulnerabilities exist in some compiler versions above 0.8.0.

References

- Ethereum Smart Contract Best Practices - Locking Pragmas
- SWC-103 - Floating Pragma
- Solidity Version Pragma

KS-ASB-07 – Library functions' first argument should be self

Severity	**Informational**
Status	**Open**

Description

The Solidity style guide recommends that the first argument should always be named `self`:

> When writing library functions that operate on a custom struct, the struct should be the first argument and should always be named `self`.

This is not the case for the functions in the HashUtils library.

References

- Solidity Style Guide - Function Argument Names

KS-ASB-08 – No emergency stop for Pool

Severity	**Informational**
Status	**Open**

Description

An emergency stop can be used to stop a smart contract in the case that a bug or vulnerability is discovered. By stopping the contract a potential attacker will not be able to exploit the vulnerability. This will leave more time to the technical staff to solve without additional loss of funds.

The Pool contract does not implement any emergency stop functionality.

References

- Ethereum Smart Contract Best Practices - Circuit Breakers
- OpenZeppelin Docs - Security: Pausable

KS-ASB-09 – No payment for publishing to Wormhole

Severity	**Informational**
Status	**Open**

Description

When the WormholeMessenger contract publishes messages through the Wormhole, no message fee is specified.

References

- GitHub: Wormhole v2.10.3 - /ethereum/contracts/Implementation.sol

KS-ASB-10 – Shadowed constant in Pool

Severity	**Informational**
Status	**Open**

Description

A child contract may introduce fields that shadows identically named fields inherited from a parent contract.

The identically named referenced in functionality implemented in the child or parent contract will refer to different state variables which may cause confusion and invalid assumptions on the behavior of the code.

The Pool contract inherits RewardsManager and both contracts implements the same constant.

References

- SWC-119 - Shadowing State Variables

KS-ASB-11 – Pragma ABI coder v2 is not experimental

Severity	**Informational**
Status	**Open**

Description

The code enables the ABI coder version 2 as an experimental feature.

Since Solidity version 0.6.0 the ABI coder version 2 is no longer considered an experimental feature. Furthermore, since Solidity version 0.8.0 the ABI coder version 2 is enabled by default.

References

- Solidity ABI Coder Pragma

KS-ASB-12 – Pragma solidity versions are inconsistent

Severity	**Informational**
Status	**Open**

Description

Different Solidity versions are defined in different files.

KS-ASB-13 – Reentrancy from ERC20 contract during Pool deposit

Severity	**Informational**
Status	**Open**

Description

When calling external contracts there is a possibility that the contract will call back. If state changes are not made until the external call returns, then a reentrancy attack can occur.

The `deposit` function in the Pool contract call the ERC20 contract to transfer tokens from the user. After the token transfer returns, internal Pool state is updated. The ERC20 contract could use this for a reentrancy attack.

References

- Ethereum Smart Contract Best Practices: External Calls: Avoid state changes after external calls
- SWC-107: Reentrancy

KS-ASB-14 – Shadowed constant in Pool

Severity	**Informational**
Status	**Open**

Description

A child contract may introduce fields that shadows identically named fields inherited from a parent contract.

The identically named referenced in functionality implemented in the child or parent contract will refer to different state variables which may cause confusion and invalid assumptions on the behavior of the code.

The Pool contract inherits RewardsManager and both contracts implements the same constant.

References

- SWC-119: Shadowing State Variables

KS-ASB-15 – Test suite excludes all but one test

Severity	**Informational**
Status	**Open**

Description

Test execution only executes one out of 43 tests. The reason that a single test is marked to be executed exclusively causing all other tests to be ignored…

References

- Mocha: Exclusive tests
- Hardhat: Running tests in parallel

METHODOLOGY

During this source code review, the Kudelski Security Services team reviewed code within the project within an appropriate IDE. During every review, the team spends considerable time working with the client to determine correct and expected functionality, business logic, and content to ensure that findings incorporate this business logic into each description and impact. Following this discovery phase the team works through the following categories:

- Authentication
- Authorization and Access Control
- Auditing and Logging
- Injection and Tampering
- Configuration Issues
- Logic Flaws
- Cryptography

These categories incorporate common vulnerabilities such as the OWASP Top 10

Tools

The following tools were used during this portion of the test. A link for more information about the tool is provided as well.

- Visual Studio 2019
- Visual Studio 2022
- Visual Studio Code
- Semgrep
- SonarQube

Vulnerability Scoring Systems

Kudelski Security utilizes a vulnerability scoring system based on impact of the vulnerability, likelihood of an attack against the vulnerability, and the difficulty of executing an attack against the vulnerability based on a high, medium, and low rating system

Impact
The overall effect of the vulnerability against the system or organization based on the areas of concern or affected components discussed with the client during the scoping of the engagement.

High:
The vulnerability has a severe affect on the company and systems or has an affect within one of the primary areas of concern noted by the client

Medium:
It is reasonable to assume that the vulnerability would have a measurable affect on the company and systems that may cause minor financial or reputational damage.

Low:
There is little to no affect from the vulnerability being compromised. These vulnerabilities could lead to complex attacks or create footholds used in more severe attacks.

Likelihood
The likelihood of an attacker discovering a vulnerability, exploiting it, and obtaining a foothold varies based on a variety of factors including compensating controls, location of the application, availability of commonly used exploits, and institutional knowledge

High:
It is extremely likely that this vulnerability will be discovered and abused

Medium:
It is likely that this vulnerability will be discovered and abused by a skilled attacker

Low:
It is unlikely that this vulnerability will be discovered or abused when discovered.

Difficulty
Difficulty is measured according to the ease of exploit by an attacker based on availability of readily available exploits, knowledge of the system, and complexity of attack. It should be noted that a LOW difficulty results in a HIGHER severity.

Low:
The vulnerability is easy to exploit or has readily available techniques for exploit

Medium:
The vulnerability is partially defended against, difficult to exploit, or requires a skilled attacker to exploit.

High:
The vulnerability is difficult to exploit and requires advanced knowledge from a skilled attacker to write an exploit

Severity
Severity is the overall score of the weakness or vulnerability as it is measured from Impact, Likelihood, and Difficulty.