Security assessment and code review

December 17 , 2021
Updated: Jan 4, 2022

Prepared for:
Ping Chen | Hakka Finance

Prepared by:
Er-Cheng Tang | HashCloak Inc
Thomas Steeger | HashCloak Inc

Table Of Contents

Executive Summary

Hakka Finance engaged HashCloak Inc for an audit of their iGain v2 smart contracts, written in Solidity. The audit was done with two auditors over a 2 week period.
The scope of the audit were solidity contracts in the v2 folder of the iGain codebase at commit hash 39ae2674c7c2dac3b6db93718a48383f1a4a9e63. There are additional tests covered in a separate branch with commit hash efa1102a6fe56faee84732ecc480a95056b54d7f.

During the first week, we familiarized ourselves with the iGain v2 smart contracts and started our manual analysis of the smart contracts. During the second week of the audit, we ran automated analysis tools on all the contracts in the iGain v2 code base. After the initial report was delivered, the Hakka Finance team updated their codebase. The new commit is bed19965c8c5b707ec80d641e069b073964d9a2a. We updated the status of our findings.

We identified issues ranging from low severity to informational and provided recommendations to improve code quality and mitigations against several attacks. We also provide some general recommendations for further improving the code base.

Severity	**Number of Findings**
Critical	0
High	0
Medium	0
Low	1
Informational	1

Overview

The Hakka Finance iGain is a financial protocol that provides option creation and trading by transforming base tokens into long and short tokens. Users can trade these tokens either on Hakka Finance iGain or any place that supports ERC20 trading. After expiration time, the separate token prices will be settled down in terms of the base token, according to certain price changes or interest rate changes.

The original idea in v1 was to give traders the opportunity to hedge against impermanent loss by creating tokens which settle at closing time to the impermanent loss of a pool. The v2 version abstracts away the underlying functionality of v1 by introducing a base contract IGainBase which provides all the necessary functionality. The solidity version was also updated and corresponding changes of the code base were made. Other contracts like IGainIL or IGainAAVEIRS introduce different features and inherit the base functionality from IGainBase. The contract TokenFactory provides the ERC20 functionality needed for the long and short tokens.

Findings

Return values of external calls should be checked

Type: Low Severity
Files affected: IGainAAVEIRS.sol, IGainYearnIRS.sol

Without control over external contracts, developers have to pay extra attention to their behaviors when building applications on them. Both IGainAAVEIRS and IGainYearnIRS will fetch the initial rate/price from external contracts upon initialization. The return values should be validated to be non-zero before ending the `init()` function. Otherwise, future calls to `close()` will always be reverted because of division by the zero initial rate/price. This breaks the mechanism for price finalization.

Impact: Long and short token prices might not be tied to the underlying rate change of an asset. Participants who want to quit have to withdraw tokens instead of being able to claim tokens, which incurs an additional fee.
Suggestion: Add `require(initialRate > 0)` at the end of `init()`.

Status: Fixed at bed19965c8c5b707ec80d641e069b073964d9a2a.

Incomplete abstraction of block timestamp

Type: Informational
Files affected: IGainAAVEIRS.sol, IGainDelta.sol, IGainIL.sol, IGainYearnIRS.sol

To abstract the implementation of getting block timestamp, a corresponding function `_blockTimestamp()` was defined. However, some parts of the code still use the actual implementation instead of calling `_blockTimestamp()`.

Suggestion: Replace `block.timestamp` with `_blockTimestamp()`.
Status: Fixed at bed19965c8c5b707ec80d641e069b073964d9a2a.

General Recommendations

Use import statements to avoid code bloat

Codes from external libraries are placed directly in the main contract, which appears cumbersome. We suggest putting libraries into separate files and importing the libraries in the main contract to improve code readability.

Validate the addresses

We recommend checking that input addresses are non-zero to prevent possible errors regarding uninitialized addresses. We also recommend validating that input addresses are indeed contracts whenever it should be the case.

Make use of function modifiers

We suggest abstracting duplicated code segments into modifiers. This could reduce errors and simplify code maintenance.