Security assessment and code review

Prepared for:
Ping Chen | Hakka Finance and Pelith

Prepared by:
Er-Cheng Tang | HashCloak Inc
Mikerah Quintyne-Collins | HashCloak Inc

Initial Delivery: March 22, 2021
Updated: April 12, 2021
Camera-Ready: April 23, 2021

Table Of Contents

Executive Summary

Hakka Finance engaged HashCloak Inc for an audit of their Urban Giggle smart contracts, written in Solidity. The audit was done with two auditors over a 2 week period. Followed by, one week of review after our recommendations and findings were applied.

During the first week, we familiarized ourselves with the Urban Giggle smart contracts and started our manual analysis of the smart contracts. During the second week of the audit, we ran several automated analysis tools (Mythril and Slither) on all the contracts in the Urban Giggle repository. Further, we did a mathematical check of the iGain protocols documentation and formulas.

We identified several issues ranging from High to Informational and provided recommendations to improve code quality and mitigations against several attacks.

Severity	**Number of Findings**
Critical	0
High	1
Medium	1
Low	1
Informational	7

Scope

The Urban Giggle codebase was assessed at commit 3e6e13e8997e2524cf0da96f6a49903a3eb849a7 between March 8 and March 19, 2021.

Fixes were applied and assessed through several commits from 1a8c3a3fc7976e9f5a574e171d15f605954ee3ce to

ebb2c0957ee4379f444dbe51859e39b1ab7de9b3 between March 26 and April 23, 2021

All the files ending in .sol in the codebase were considered.

Overview

The Urban Giggle repository is comprised of several self-contained smart contracts for the following Hakka Finance products:

- Thank You Token: A proof of donation token forked and modified from the reference WETH smart contract
- Vesting Vault: A vesting schedule voted by the Hakka Finance community in HIP20 and HIP33
- Vesting Reward Contract: A fork of an older Synthetix LP vesting rewards contract
- Hakka Intelligence: A prediction market for betting on price changes of different assets.
- ImpermanentGain: A market that offers hedging against impermanent loss which AMM liquidity providers are concerned about

Findings

Return values of external calls should be checked

Type: High

Description: Throughout the codebase, it is noted that return values of calls to external contracts are not checked. This can lead to unexpected behavior in the contract as the determination of these values are out of control and may turn out to be exploitable.

This issue has a severe consequence in the Hakka Intelligence contract. In particular, a transaction for the first call to `proceed()` will succeed even if an oracle returns zero. This value is assigned to the contract storage once and for all, and in later transactions it will act as a denominator. This logic causes the contract to be blocked afterwards because subsequently calling `proceed()` will encounter a zero denominator and fail. The impact is that all bets are permanently locked in the contract in this case because the contract can never have the answer set. This problem could be avoided if oracle return values were validated beforehand.

Status: The team made a fix at commit [1a8c3a3fc7976e9f5a574e171d15f605954ee3ce](1a8c3a3fc7976e9f5a574e171d15f605954ee3ce).
This issue is properly resolved to prevent unexpected or malicious behavior of this kind.

Arbitrage opportunity exists under public information and linear reward

Type: Medium

Description: In Hakka Intelligence, due to the fact that submissions are public, it is possible to take advantage of other user's submissions. A sophisticated user would like to collect all available submissions (possibly including those in mempool) and choose to make a submission right before the countdown ends. We observed that the linear reward mechanism indeed facilitated a strategy to perform arbitrage, as detailed in Appendix A. Although this does not affect the other parties' chances of winning, it does affect the actual payoff and the fairness of Hakka Intelligence, thus hurting incentives for participants to submit based on their real views.

Status: The team made fixes at commit [1a8c3a3fc7976e9f5a574e171d15f605954ee3ce](1a8c3a3fc7976e9f5a574e171d15f605954ee3ce) to modify the reward function, and at [b447dcd985a42a538bd4796347bc233a63c45b61](b447dcd985a42a538bd4796347bc233a63c45b61) to accept only private submissions.
This issue is properly resolved by both assessments. We provide an analysis of the new reward formula in appendix B.

Functions likely to fail under normal usage

Type: Low

In the iGain contract, the function `burnLP()` is likely to fail for many ERC20 tokens when they have a total supply exceeding 10^{18}, which is often the case. This might be an unexpected situation from the users' point of view. To be more precise, notice that four potentially large numbers are multiplied together in line 416 of `burnLP()`. Although integer overflow is protected using a safe math library, the transaction will still be reverted if the multiplicands are too large. One can fix this problem by altering the order of calculation. We suggest that `(fee()*lp/_totalSupply)` is calculated first. This way, `burnLP()` can function normally as long as the total supply is under 10^{27}, while still getting accurate enough numeric results.

Status: The team made commits at [adb72d26230d12f23b21dd9506dc978c7c150bc1] and [9683c2a8bd588a25ea8f74005b25c7f8a117eb58] to fix the issue in two ways. It can be seen as a tradeoff between arithmetic precision and applicability plus gas efficiency. We provide a comparison in appendix C for reference.

Formulas in the documentation are inconsistent with the implementation

Type: Informational

Description:
There are inconsistencies between the documentation and implementation of the iGain contract. The initial relationship for `burnA` in the current documentation should be $A * B = (A + fx)(B - y)$ instead of $A * B = (A - fx)(B + y)$. The resulting formula $x = \frac{af - A - Bf + \sqrt{(f(B-a)+A)^2 - 4aAf}}{2f}$ also contains a typo, where `-4aAf` should be replaced with `+4aAf`.

Status: The document is properly fixed to avoid confusion.

No descriptive error messages upon error

Type: Informational

Description: In several contracts, the usage of `require()` and `revert()` do not provide description error messages in the case of failure.

Status: The team made a fix at commit [1a8c3a3fc7976e9f5a574e171d15f605954ee3ce].
This issue is properly resolved to enhance user experience.

No unit tests throughout Urban Giggle codebase

Type: Informational

There are no unit tests in the Urban Giggle repository for ensuring the intended behavior and usage of the contracts.

Unchecked withdrawal

Type: Informational

Description: During the withdrawal of funds from the VestingVault contract, it is noted that addresses that didn't deposit funds into the vault can call the `withdraw` function. Although due to uint256 defaulting to 0, the user would withdraw no funds, it wastes gas to execute the entire function given that the user didn't participate.

Usage of `now()` instead of `block.timestamp()`

Type: Informational

Description: The `now()` keyword in Solidity has been deprecated since version 0.7.0.

Usage of block values as a proxy for time

Type: Informational

Description: Due to the use of the current block.timestamp to delineate start and ending of events, it is possible for miners to manipulate the start and end times for the events. This can affect fairness for users as they are expecting events to start and end at certain times.

Missing zero address validation of parameters

Type: Informational

Description: In `setRewardDistribution()` of the HakkaRewardsVesting contract ,the null address is not accounted for. It is possible to call this function with the null address, potentially causing an unintended transition of privileged rights, although the contract owner can make another transaction to fix that in such cases.

Gas usage can be optimized

Type: Informational

Description: In the iGain contract, several optimizations on gas usage can be made. For example, the number of multiplications can be reduced by rewriting

```
        _fee = closeTime.sub(time).mul(minFee).add(
                    time.sub(openTime).mul(maxFee)
                ).div(closeTime.sub(openTime));
```

as

```
        _fee = minFee.add(
                    time.sub(openTime).mul((maxFee.sub(minFee))).div(closeTime.sub(openTime))
                );
```

As for the implementation of `burnA`, the original code snippet is

```
x = reserveOut.sub(amountIn).mul(f).div(1e18).add(reserveIn);
x = x.mul(x).add(amountIn.mul(4).mul(reserveIn).mul(f).div(1e18)).sqrt();
x = x.add(amountIn.mul(f).div(1e18)).sub(reserveOut.mul(f).div(1e18)).sub(reserveIn);
```

But most terms in the last line were already covered by the first line, so we could reuse the result and simply write

```
x = reserveOut.sub(amountIn).mul(f).div(1e18).add(reserveIn);
x = x.mul(x).add(amountIn.mul(4).mul(reserveIn).mul(f).div(1e18)).sqrt().sub(x);
```

Status: The team made commits [adb72d26230d12f23b21dd9506dc978c7c150bc1](adb72d26230d12f23b21dd9506dc978c7c150bc1) and [5859c377f5cf7f751a5f26f1689742210e5530cd](5859c377f5cf7f751a5f26f1689742210e5530cd) to apply the optimizations.

Recommendations

Update to more recent stable of solidity

In order to leverage the added features with regards to correctness and stability, we recommend that all the contracts are updated to use a more recent and stable version of stability.

Add unit tests

In order to ensure the correct intended behavior and usage of the Urban Giggle contracts, we highly recommend that unit tests are written.

Update now() to block.timestamp()

Due to its deprecation in Solidity 0.7.0, we recommend that all usage of the `now()` be updated to `block.timestamp()`.

Ensure that events can maintain their integrity if the timestamp is delayed

Due to the use of time dependent block variables, we highly recommend that any events that do require such functionality be minimally affected in terms of integrity and execution of the protocol.

Validate addresses to safeguard against the null address

In order to minimize the potential of unintended behavior due to a user calling `setRewardDistribution()` with the null address, we recommend that the function validates the address and revert upon encountering the null address.

Use commit-reveal to improve fairness

In Hakka Intelligence, due to the openness of users' submissions, we recommend that a commit and reveal scheme be integrated in order to improve the fairness of the protocol.

Correct the iGain documentation

We recommend that the documentation is updated with the correct formula derivation for a complete exposition.

Optimize gas usage

Due to costly gas fees, we recommend making changes on the current implementation to perform calculations in more gas-efficient ways as pointed out.

Ensure the consistency of comments

Due to code readability, comments should be consistent with the implementation. In particular there should be no `sqrt()` involved in the mentioned comments in lines 400, 466, 485 of the iGain contract.

Appendix A: Arbitrage for Hakka Intelligence

Say there are n stakes c_i with unit-vector submissions $\vec{v_i}$ arranged according to transaction order. Let $\vec{u}$ be the normalized unit vector of $\Sigma_{i=1}^n c_i \vec{v_i}$. Let's examine what the n-th submitter can earn by choosing $\vec{v_n} = \vec{u}$

For any unit-vector answer a, the sum of scores that are revealed is

$$score_{reveal} \leq score_{all} = \Sigma_{i=1}^n c_i \langle \vec{a}, \vec{v_i} \rangle = \langle \vec{a}, \Sigma_{i=1}^n c_i \vec{v_i} \rangle = \langle \vec{a}, \vec{u} \rangle ||\Sigma_{i=1}^n c_i \vec{v_i}|| \leq \langle \vec{a}, \vec{u} \rangle (\Sigma_{i=1}^n c_i)$$

where the latter inequality comes from the triangle inequality. It will be a strict inequality if there are distinct submissions, which is likely the case in this application.
The last submitter gets reward

$$R_n = \frac{score_n}{score_{reveal}} \cdot stake_{all} = \frac{c_n \langle \vec{a}, \vec{u} \rangle}{score_{reveal}} \cdot (\Sigma_{i=1}^n c_i) \geq \frac{c_n \langle \vec{a}, \vec{u} \rangle}{\langle \vec{a}, \vec{u} \rangle (\Sigma_{i=1}^n c_i)} \cdot (\Sigma_{i=1}^n c_i) = c_n$$

The reward R_n is strictly more than the stake c_n if there have been distinct submissions, and this is true regardless of the outcome of the answer.

Appendix B: Justification of the Updated Reward Formula

We argue here that the updated reward formula properly mitigates the arbitrage problem.

- Say there are n stakes c_i with unit-vector submissions $\vec{v_i} = (v_i^{(1)}, \cdots, v_i^{(m)})$. If the unit-vector answer is $\vec{a}$, the i-th party will get a score $\sigma_i = c_i \langle \vec{a}, \vec{v_i} \rangle^2$ and a reward $R_i = \frac{\sigma_i}{\Sigma_k \sigma_k} (\Sigma_k c_k)$. A strict arbitrage opportunity means that $R_i > c_i$ regardless of the outcome of $\vec{a}$, which translates to the condition $\sigma_i (\Sigma_k c_k) > c_i (\Sigma_k \sigma_k)$.
- Consider the m specific answers $\vec{e_1}, \cdots, \vec{e_m}$, that is, $e_i^{(j)} = 1_{\{i=j\}}$. Denote the score and reward of the i-th party for answer $\vec{a} = \vec{e_j}$ as $\sigma_i^{(j)}$ and $R_i^{(j)}$. Then we have $\sigma_i^{(j)} = c_i \langle \vec{e_j}, v_i \rangle^2 = c_i (v_i^{(j)})^2$. This also implies that $c_i = c_i ||v_i||^2 = \Sigma_j c_i (v_i^{(j)})^2 = \Sigma_j \sigma_i^{(j)}$.
- Combining the strict arbitrage condition of party i over answers $\vec{e_1}, \cdots, \vec{e_m}$, we get $\Sigma_j (\sigma_i^{(j)} (\Sigma_k c_k)) > \Sigma_j (c_i (\Sigma_k \sigma_k^{(j)}))$. So $(\Sigma_j \sigma_i^{(j)})(\Sigma_k c_k) > c_i (\Sigma_k (\Sigma_j \sigma_k^{(j)}))$, that is, $c_i (\Sigma_k c_k) > c_i (\Sigma_k c_k)$, a contradiction. This proves that strict arbitrage cannot happen.
- By relaxing the strictness requirement in our argument, we conclude that arbitrage opportunity does not exist in the current scheme. If someone can earn money for some answer, he/she may also lose in other cases.

Appendix C: Comparing Order of Calculations

The updated order of calculation for `burnLP` is

```
amount =
poolA.mul(poolB).mul(4).mul(fee().mul(lp).div(1e18)).div(_totalS
upply);
amount =
amount.mul((2e18).sub(lp.mul(fee()).div(_totalSupply))).div(1e18
);
```

There are 3 multiplicands `poolA`, `poolB` and `fee().mul(lp).div(1e18)` in the beginning that can be roughly the same scale.Overflow can be prevented as long as the total supply of base token is within 10^{24}. We have suggested another order of calculation, namely,

```
uint256 f = fee().mul(lp).div(_totalSupply)
amount = poolA.mul(poolB).mul(4).mul(f).div(1e18);
amount = amount.mul((2e18).sub(f)).div(1e18);
```

`f` is at most 10^{18}, so the multiplicands poolA and poolB will not cause an overflow as long as they are within 10^{27}. This expands applicability. Writing this way can also save calculations that would have been performed twice. However there can be more loss of precision when calculating the amount, acting as a tradeoff.