SMART CONTRACT AUDIT REPORT

for

HAKKA FINANCE

Prepared By: Shuxiao Wang

PeckShield

March 7, 2021

Document Properties

Client	Hakka Finance
Title	Smart Contract Audit Report
Target	iGain
Version	1.0
Author	Xuxian Jiang
Auditors	Ruiyi Zhang, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 7, 2021	Xuxian Jiang	Final Release
1.0-rc	March 4, 2021	Xuxian Jiang	Release Candidate
0.2	March 3, 2021	Xuxian Jiang	Additional Findings #1
0.1	February 26, 2021	Xuxian Jiang	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the iGain protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About iGain

The iGain protocol is a decentralized financial instrument that provides the options for investors to hedge, profit, or speculate on certain targeting underlying assets with a synthetic, tokenized position. Specifically, it tokenizes the `CALL`/`PUT` options of underlying assets into `LONG`/`SHORT` tokens. Then, it makes use of the AMM mechanism to create a secondary market of `LONG`/`SHORT` tokens. Investors might hedge against a certain risk or earn a profit in a period through holding `LONG`/`SHORT` tokens. With this instrument, iGain presents a unique innovation in DeFi ecosystem for investors to better control potential risks.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of iGain

Item	Description
Client	Hakka Finance
Website	http://igain.hakka.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 7, 2021

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit.

- https://github.com/artistic709/urban-giggle.git (ae75b50)

1.2 About PeckShield

PeckShield Inc. [12] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [11]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [10], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the given iGain contracts. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	3	■ ■ ■
Informational	3	■ ■ ■
Total	7	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 3 low-severity vulnerabilities, and 3 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Suggested Adherence of Checks-Effects-Interactions	Time and State	Confirmed
PVE-002	Low	Incompatibility with Deflationary/Rebasing Tokens	Business Logic	Confirmed
PVE-003	Low	Improved getReward() Logic	Business Logic	Fixed
PVE-004	Medium	Oversized Rewards May Lock All Pool Stakes	Numeric Errors	Fixed
PVE-005	Informational	Explicit Reveal Requirement in HakkaIntelligence::claim()	Coding Practices	Confirmed
PVE-006	Low	Inaccurate AddLP/RemoveLP Event Generation	Business Logic	Fixed
PVE-007	Informational	Inconsistent burnPartialHelper() Calculation From Documentation	Business Logic	Fixed

Beside the identified issues, we note that the staking support assumes the staked tokens are not deflationary. Also, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `VestingVault`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [14] exploit, and the recent `Uniswap/Lendf.Me` hack [13].

We notice an occasion where the `checks-effects-interactions` principle is violated. Using the `VestingVault` as an example, the `deposit()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 260) starts before effecting the update on internal states (line 261), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same `_deposit()` function. Note that there is no harm that may be caused to current protocol. However, it is still suggested to follow the known `checks-effects-interactions` best practice.

```
259    function deposit(address to, uint256 amount) external {
260        hakka.safeTransferFrom(msg.sender, address(this), amount);
261        balanceOf[to] = balanceOf[to].add(amount);
```

```
        emit Deposit(msg.sender, to, amount);
    }
```

Listing 3.1: VestingVault :: deposit ()

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`.

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` best practice.

Status The issue has been confirmed. The team will exercise extra caution when selecting the tokens to support in the protocol.

3.2 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `HakkaRewardsVesting`
- Category: Business Logics [7]
- CWE subcategory: CWE-841 [5]

Description

Among the audited contracts, the `HakkaRewardsVesting` contract is designed to be the main entry for interaction with staking users. In particular, one entry routine, i.e., `deposit()`, accepts user deposits of supported assets (e.g., `DAI`). Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the `HakkaRewardsVesting` contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    function stake(uint256 amount) public {
        _totalSupply = _totalSupply.add(amount);
        _balances[msg.sender] = _balances[msg.sender].add(amount);
        stakeToken.safeTransferFrom(msg.sender, address(this), amount);
    }

    function stakeFor(address to, uint256 amount) public {
        _totalSupply = _totalSupply.add(amount);
        _balances[to] = _balances[to].add(amount);
        stakeToken.safeTransferFrom(msg.sender, address(this), amount);
    }
```

Listing 3.2: HakkaRewardsVesting::stake()

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines.

One possible mitigation is to regulate the set of ERC20 tokens that are permitted into the `HakkaRewardsVesting`. In our case, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., `USDT`) that may have control switches that can be dynamically exercised to suddenly become one.

Recommendation If current codebase needs to support possible deflationary tokens, it is better to check the balance before and after the `transfer()/transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is widely-adopted `USDT`.

Status This issue has been confirmed. However, considering the fact that this specific issue does not affect the normal operation, the team decides to address it when the need of supporting deflationary/rebasing tokens arises.

3.3 Simplified Logic in getReward()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `HakkaRewardsVesting`
- Category: Business Logic [7]
- CWE subcategory: CWE-770 [4]

Description

In the `HakkaRewardsVesting` contract, the `getReward()` routine is intended to obtain the calling user's staking rewards. The logic is rather straightforward in calculating possible reward, which, if not zero, is then allocated to the calling (staking) user.

Our examination shows that the current implementation logic can be further optimized. In particular, the `getReward()` routine has a modifier, i.e., `updateReward(msg.sender)`, which timely updates the calling user's (earned) rewards in `rewards[msg.sender]` (line 573).

```
    function getReward() public updateReward(msg.sender) {
        uint256 reward = earned(msg.sender);
        if (reward > 0) {
            rewards[msg.sender] = 0;
            vault.deposit(msg.sender, reward);
```

```
639                 emit RewardPaid(msg.sender, reward);
640             }
641         }
```

Listing 3.3: HakkaRewardsVesting::getReward()

```
    modifier updateReward(address account) {
        rewardPerTokenStored = rewardPerToken();
        lastUpdateTime = lastTimeRewardApplicable();
        if (account != address(0)) {
            rewards[account] = earned(account);
            userRewardPerTokenPaid[account] = rewardPerTokenStored;
        }
        _;
    }
```

Listing 3.4: HakkaRewardsVesting::updateReward()

Having the modifier `updateReward()`, there is no need to re-calculate the earned reward for the caller `msg.sender`. In other words, we can simply re-use the calculated `rewards[msg.sender]` and assign it to the `reward` variable (line 635).

Recommendation Avoid the duplicated calculation of the caller's reward in `getReward()`, which also leads to (small) beneficial reduction of associated gas cost. An example revision is shown below.

```
634     function getReward() public updateReward(msg.sender) {
635         uint256 reward = rewards[msg.sender];
636         if (reward > 0) {
637             rewards[msg.sender] = 0;
638             vault.deposit(msg.sender, reward);
639             emit RewardPaid(msg.sender, reward);
640         }
641     }
```

Listing 3.5: HakkaRewardsVesting::getReward()

Status This issue has been fixed in the commit: bbac0e8.

3.4 Oversized Rewards May Lock All Pool Stakes

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `HakkaRewardsVesting`
- Category: Numeric Errors [9]
- CWE subcategory: CWE-190 [2]

Description

In this section, we continue to examine the `HakkaRewardsVesting` logic and focus on the `rewardPerToken()` routine. This routine is responsible for calculating the reward rate for each staked token and it is part of the `updateReward` modifier that would be invoked up-front for almost every public function in `HakkaRewardsVesting` to update and use the latest reward rate.

Our analysis leads to the discovery of a potential pitfall when a new oversized reward amount is added into the pool. In particular, as the `rewardPerToken()` routine involves the multiplication of three `uint256` integer, it is possible for their multiplication to have an undesirable overflow (lines 593−599), especially when the `rewardRate` is largely controlled by an external entity, i.e., `rewardDistribution` (through the `notifyRewardAmount()` function).

```
569     modifier updateReward(address account) {
570         rewardPerTokenStored = rewardPerToken();
571         lastUpdateTime = lastTimeRewardApplicable();
572         if (account != address(0)) {
573             rewards[account] = earned(account);
574             userRewardPerTokenPaid[account] = rewardPerTokenStored;
575         }
576         _;
577     }

579     constructor(IERC20 _stakeToken) public {
580         stakeToken = _stakeToken;
581         hakka.safeApprove(address(vault), uint256(-1));
582     }

584     function lastTimeRewardApplicable() public view returns (uint256) {
585         return Math.min(block.timestamp, periodFinish);
586     }

588     function rewardPerToken() public view returns (uint256) {
589         if (totalSupply() == 0) {
590             return rewardPerTokenStored;
591         }
592         return
593             rewardPerTokenStored.add(
594                 lastTimeRewardApplicable()
```

```
                        .sub(lastUpdateTime)
                        .mul(rewardRate)
                        .mul(1e18)
                        .div(totalSupply())
                );
    }
```

Listing 3.6: HakkaRewardsVesting::updateReward()

This issue is made possible if the reward amount is given as the argument to `notifyRewardAmount()` such that the calculation of `rewardRate.mul(1e18)` always overflows, hence locking all deposited funds. Note that an authentication check on the caller of `notifyRewardAmount()` greatly alleviates such concern. Currently, only the `rewardDistribution` address is able to call `notifyRewardAmount()` and this address is set by the owner. Apparently, if the owner is a normal address, it may put users' funds at risk. To mitigate this issue, it is important to transfer the ownership to the governance and ensure the given reward amount will not be oversized to overflow and lock users' funds.

Recommendation Ensure the reward amount is appropriate, without resulting in overflowing and locking users' funds.

Status This issue has been fixed in the commit: bbac0e8.

3.5 Explicit Reveal Requirement in HakkaIntelligence::claim()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `HakkaIntelligence`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

The `HakkaIntelligence` contract implements a prediction market-like contract to bet on price changes. Players can make their submissions within the allowed bet period and then reveal their results afterwards to calculate their scores. The protocol calculates the share based on the revealed scores and give out the bet rewards. After that, players can then claim their rewards. In the following, we examine the `claim()` logic.

To elaborate, we show below the `claim()` routine. It implements a rather straightforward logic by computing and distributing the reward share (line 233). However, it will be helpful to ensure the claiming user has already revealed to get their scores. In other words, in addition to current two requirements (lines $230-231$), it is helpful to add a third requirement, i.e., `require(player.reveal)`. Otherwise, a malicious actor may attempt to modify the `claimed` state of a player even before the bet begins. Fortunately, the share calculation requires the `totalScore.sub(offset)` as the denominator, which could revert the execution.

```
    function claim(address _player) public returns (uint256 amount) {
        Player storage player = players[_player];

        require(now > revealClose);
        require(!player.claimed);
        player.claimed = true;
        amount = token.balanceOf(address(this)).mul(player.score).div(totalScore.sub(
            offset));
        offset = offset.add(player.score);
        token.safeTransfer(_player, amount);

        emit Claim(_player, amount);
    }
```

Listing 3.7: HakkaIntelligence :: claim()

Recommendation Add the additional requirement on the reveal state of the claiming player. An example revision is shown below:

```
    function claim(address _player) public returns (uint256 amount) {
        Player storage player = players[_player];

        require(now > revealClose);
        require(!player.claimed);
        require(player.reveal);
        player.claimed = true;
        amount = token.balanceOf(address(this)).mul(player.score).div(totalScore.sub(
            offset));
        offset = offset.add(player.score);
        token.safeTransfer(_player, amount);

        emit Claim(_player, amount);
    }
```

Listing 3.8: HakkaIntelligence :: claim()

Status The team has confirmed that it is a design choice.

3.6 Inaccurate AddLP/RemoveLP Event Generation

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ImpermanentGain`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [5]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `ImpermanentGain` contract as an example. This contract is designed to tokenize the impermanent loss. The tokenized impermanent loss can therefore be traded and swapped. While examining the events that reflect the `ImpermanentGain` dynamics, we notice the emitted `AddLP` event (line 277) contains incorrect information. Specifically, the event is defined as `event AddLP(address indexed provider, uint256 a, uint256 b, uint256 lp);` with a number of parameters: the first parameter `provider` encodes the address that performs the LP-adding operation; the second and third parameters show the `LONG/SHORT` token amounts; while the last indicates the LP amount. The emitted event contains an incorrect order as the last parameter is actually in the second parameter.

```
248     function init(address _baseToken, address _oracle, address _treasury, uint256
            _duration, uint256 _a, uint256 _b) public {
249         require(openTime == 0, "Initialized");
250         require(_a > 0 && _b > 0, "No initial liquidity");
251         baseToken = _baseToken;
252         oracle = Oracle(_oracle);
253         treasury = _treasury;
254         openTime = now;
255         closeTime = now.add(_duration);
256         openPrice = uint256(oracle.latestAnswer());
257 
258         canBuy = true;
259 
260         name = "iGain LP token";
261         symbol = "iGLP";
262         decimals = ERC20Mintable(baseToken).decimals();
263 
264         uint256 _lp = _a.mul(_b).sqrt();
265         poolA = _a;
```

```
        poolB = _b;
        _mint(msg.sender, _lp);
        _mint(address(0), 1000); //lock liquidity
        if(_b > _a) {
            a[msg.sender] = _b.sub(_a);
            doTransferIn(baseToken, msg.sender, _b);
        }
        else {
            b[msg.sender] = _a.sub(_b);
            doTransferIn(baseToken, msg.sender, _a);
        }
        emit AddLP(msg.sender, _lp, _a, _b);
    }
```

Listing 3.9: ImpermanentGain::init()

Note the same issue is also applicable to the `RemoveLP` events.

Recommendation Properly emit the `AddLP` event with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status This issue has been fixed in the commit: 3e6e13e.

3.7 Inconsistent burnPartialHelper() Calculation From Documentation

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `ImpermanentGain`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [5]

Description

The `ImpermanentGain` contract provides a number of helper routines that facilitate the conversion or swapping of related tokens, i.e., `baseToken`, `a`, and `b`. One specific helper routine `burnPartialHelper()` is used to calculate how many of `a` needs to be swapped for `b` when burning `a`.

We have examined the formula behind this routine's logic and notices the calculation in the implementation is inconsistent from the given documentation. Specifically, the formula behind the correct computation should be: $x = \frac{\sqrt{(f(B-a)+A)^2+4aAf}+af-A-Bf}{2f}$. However, the given document shows the following: $x = \frac{\sqrt{(f(a+B)+A)^2-4aAf}+af+A+Bf}{2f}$.

```
    // calculate how many of a needs to be swapped for b when burning a
```

```
    function burnPartialHelper(uint256 amountIn, uint256 reserveIn, uint256 reserveOut,
        uint256 f) internal pure returns (uint256 x) {
        x = reserveOut.sub(amountIn).mul(f).div(1e18).add(reserveIn); // (reserveOut - a
            ) * fee + 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);
        x = x.mul(1e18).div(f).div(2);
    }
```

Listing 3.10: ImpermanentGain::burnPartialHelper()

Recommendation Make the documentation consistent with the current implementation regarding the `burnPartialHelper()` routine.

Status This issue has been fixed by updating the documentation located in the following link: burnA().

4 Conclusion

In this audit, we have analyzed the iGain design and implementation. The system presents a unique offering in DeFi ecosystem for investors to better control potential risks. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[4] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.html.

[5] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[9] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[10] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[11] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[12] PeckShield. PeckShield Inc. https://www.peckshield.com.

[13] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[14] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.