SMART CONTRACT AUDIT REPORT

for

HAKKA FINANCE

Prepared By: Shuxiao Wang

Hangzhou, China

September 18, 2020

Document Properties

Client	Hakka Finance
Title	Smart Contract Audit Report
Target	BlackHoleSwap
Version	1.0-patch1
Author	Xuxian Jiang
Auditors	Huaguo Shi, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-patch1	September 18, 2020	Xuxian Jiang	Fix the github link in Section 1.1
1.0	August 15, 2020	Xuxian Jiang	Final Release
1.0-rc1	August 10, 2020	Xuxian Jiang	Additional Findings
0.1	August 8, 2020	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 **BlackHoleSwap** smart contract source code, we in the report outline 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 contract can be further improved due to the presence of several issues. This document outlines our audit results.

1.1 About BlackHoleSwap

BlackHoleSwap is a decentralized, stablecoin-oriented AMM (Automatic Market Making) exchange. It uniquely integrates with mainstream lending protocols to leverage the excess supply while borrowing on the inadequate side. By doing so, it can effectively process transactions far exceeding its existing liquidity and thus provide nearly infinite liquidity with the very low price slippage and high capital utilization. BlackHoleSwap advances the current DEX frontline and is considered a true innovation in the rapidly-evolving DeFi ecosystem.

The basic information of BlackHoleSwap is as follows:

Table 1.1: Basic Information of BlackHoleSwap

Item	Description
Issuer	Hakka Finance
Website	https://hakka.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 18, 2020

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. We note that BlackHoleSwap does not require an oracle for price feeds, but seamlessly integrates with Compound for its protocol-wide operations.

- https://github.com/artistic709/Hakka_Audit (7c03004)

1.2 About PeckShield

PeckShield Inc. [21] 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 [16]:

- 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 classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

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

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

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) [15], 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.

1.4 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), 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 an 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 BlackHoleSwap implementation. During the first phase of our audit, we studied the smart contract source code and ran 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	2	■ ■
Low	5	■ ■ ■ ■ ■
Informational	3	■ ■ ■
Total	10	

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 2 medium-severity vulnerabilities, 5 low-severity vulnerabilities, and 3 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	External Declaration of Only-Externally-Invoked Functions	Coding Practices	Fixed
PVE-002	Low	Improperly Handled Corner Cases in SafeMath	Security Features	Fixed
PVE-003	Low	Improved Precision Calculation in Multiplication and Division	Coding Practices	Fixed
PVE-004	Medium	Improved Precision Calculation With divCeil()	Coding Practices	Fixed
PVE-005	Informational	Enriched Event Generation	Time and State	Fixed
PVE-006	Medium	Safety Checks in Liquidity Addition and Removal	Time and State	Fixed
PVE-007	Low	Recommended Reentrancy Protection	Concurrency	Confirmed
PVE-008	Low	Possible Integer Overflow in *sqrt()*	Numeric Errors	Fixed
PVE-009	Low	approve()/transferFrom() Race Condition	Time and State	Confirmed
PVE-010	Informational	Better Handling of Ownership Transfer	Security Features	Confirmed

Please refer to Section 3 for details.

3 | Detailed Results

3.1 External Declaration of Only-Externally-Invoked Functions

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: blackholeswap
- Category: Coding Practices [12]
- CWE subcategory: CWE-287 [6]

Description

The BlackHoleSwap contracts provide a number of interface functions that are designed to be called only for external users. Many of these functions are defined as `public`. In `public` functions, Solidity immediately copies array arguments to memory, while `external` functions can read directly from `calldata`. Note that memory allocation can be expensive, whereas reading from calldata is not. So when these functions are not used within the contract, it's always suggested to define them as `external` instead of `public`. After analyzing the code, we recommend changing the following functions from `public` to `external`:

```
    function S() public returns (uint256)
    function dai2usdcIn(uint256 input, uint256 min_output, uint256 deadline) public
        returns (uint256)
    function usdc2daiIn(uint256 input, uint256 min_output, uint256 deadline) public
        returns (uint256)
    function dai2usdcOut(uint256 max_input, uint256 output, uint256 deadline) public
        returns (uint256)
    function usdc2daiOut(uint256 max_input, uint256 output, uint256 deadline) public
        returns (uint256)
    function addLiquidity(uint256 share, uint256[2] memory tokens) public returns (
        uint256, uint256)
    function removeLiquidity(uint256 share, uint256[2] memory tokens) public returns (
        uint256, uint256)
```

Listing 3.1: blackholeswap

Recommendation Revise the above functions from being `public` to `external`.

3.2 Improperly Handled Corner Cases in SafeMath

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SafeMath`
- Category: Security Features [10]
- CWE subcategory: CWE-284 [5]

Description

`SafeMath` is a Solidity `math` library especially designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. BlackHoleSwap encloses its own implementation by extending the support of `int256`.

We have analyzed the 9 operations defined in the library, i.e., `add/sub/mul/div/mod` for `uint256` and `add/sub/mul/div` for `int256`. Our analysis shows that one particular operation, i.e., `div` for `int256`, can be improved by better handling a subtle corner case.

Specifically, we show below the related `div` code snippet. Within the routine, there is a `require` statement that basically performs sanity checks on legitimate arguments regarding the two operands `a` and `b`. It disallows there cases: `b != 0`, `b != -1`, and `a != INT256_MIN`. The first case is reasonable while the last two cases apparently rule out legitimate cases. One such example is `a = 1` and `b = -1`.

```
function div(int256 a, int256 b) internal pure returns (int256) {
    require(b != 0 && b != -1 && a != INT256_MIN);
    int256 c = a / b;
    return c;
}
```

Listing 3.2: blackholeswapV1.sol

To avoid blocking legitimate inputs, the `require` statement can be revised to only disallow two cases: `b != 0` and `b != -1 || a != INT256_MIN`. In other words, it becomes the following: `require(b != 0 && (b != -1 || a != INT256_MIN))`.

Recommendation Revise the `div` operation to not block legitimate cases.

```
function div(int256 a, int256 b) internal pure returns (int256) {
    require(b != 0 && (b != -1  a != INT256_MIN));
    int256 c = a / b;
    return c;
}
```

Listing 3.3: blackholeswapV1.sol

3.3 Improved Precision Calculation in Multiplication and Division

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: blackholeswap
- Category: Coding Practices [12]
- CWE subcategory: CWE-627 [8]

Description

As discussed earlier, `SafeMath` is a Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss.

Using the `calcFee()` function as an example, the protocol fee charged for each trade is calculated with a combination of `add/sub/mul/div/mod` operations. All these operations are intended for `uint256`. We point out that if there is a sequence of multiplication and division operations, it is always better to calculate the multiplication before the division. By doing so, we can achieve better precision.

```
    function calcFee(uint256 input, uint256 a, uint256 b, uint256 c, uint256 d) internal {
        if(protocolFee > 0) {
            uint256 _fee = input.mul(protocolFee).div(1e18).mul(_totalSupply).div( a.add(c).sub(b).sub(d) );
            _mint(vault, _fee);
        }
    }
```

Listing 3.4: blackholeswap.sol

With that, we can develop an improved version of `calcFee()` as follows:

```
    function calcFee(uint256 input, uint256 a, uint256 b, uint256 c, uint256 d) internal {
        if(protocolFee > 0) {
            uint256 _fee = input.mul(protocolFee).mul(_totalSupply).div(1e18).div( a.add(c).sub(b).sub(d) );
            _mint(vault, _fee);
        }
    }
```

Listing 3.5: blackholeswap.sol

A further examination of BlackHoleSwap shows there exist another occasion, i.e., the `usdc2daiIn()` function. This function contains the calculation – `input.mul(fee).div(1e18).mul(rate())` that can be revised to become `input.mul(fee).mul(rate()).div(1e18)`, which can be further simplified as `input.mul(fee).div(1e6)`.

```
    function usdc2daiIn(uint256 input, uint256 min_output, uint256 deadline) public returns (uint256) {
        require(block.timestamp <= deadline, "EXPIRED");
        (uint256 a, uint256 b) = getDaiBalance();
        (uint256 c, uint256 d) = getUSDCBalance();

        uint256 output = getInputPrice(input.mul(fee).div(1e18).mul(rate()), c, d, a, b);
        securityCheck(input, output, c, d, a, b);
        require(output >= min_output, "SLIPPAGE_DETECTED");

        calcFee(input.mul(rate()), a, b, c, d);

        doTransferIn(USDC, cUSDC, d.div(rate()), msg.sender, input);
        doTransferOut(Dai, cDai, a, msg.sender, output);

        emit Purchases(msg.sender, address(Dai), input, output);

        return output;
    }
```

Listing 3.6: blackholeswap.sol

Recommendation Revise the above calculations to better mitigate possible precision loss.

```
    function usdc2daiIn(uint256 input, uint256 min_output, uint256 deadline) public returns (uint256) {
        require(block.timestamp <= deadline, "EXPIRED");
        (uint256 a, uint256 b) = getDaiBalance();
        (uint256 c, uint256 d) = getUSDCBalance();

        uint256 output = getInputPrice(input.mul(fee).div(1e6), c, d, a, b);
        securityCheck(input, output, c, d, a, b);
        require(output >= min_output, "SLIPPAGE_DETECTED");

        calcFee(input.mul(rate()), a, b, c, d);

        doTransferIn(USDC, cUSDC, d.div(rate()), msg.sender, input);
        doTransferOut(Dai, cDai, a, msg.sender, output);

        emit Purchases(msg.sender, address(Dai), input, output);

        return output;
    }
```

Listing 3.7: blackholeswap.sol (revised)

3.4 Improved Precision Calculation With divCeil()

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `blackholeswap`
- Category: Coding Practices [12]
- CWE subcategory: CWE-627 [8]

Description

In the previous section, we examined one specific source of precision loss, i.e., the order of multiplication and division operations. In this section, we examine another possible source that comes from the default division behavior, i.e., the `floor` division.

Conceptually, the `floor` division is a normal division operation except it returns the largest possible integer that is either less than or equal to the normal division result. In `SafeMath`, `floor(x)` or simply `div` takes as input an integer number x and gives as output the greatest integer less than or equal to x, denoted `floor(x)` $= \lfloor x \rfloor$. Its counterpart is the `ceiling` division that maps x to the least integer greater than or equal to x, denoted as `ceil(x)` $= \lceil x \rceil$. In essence, the `ceiling` division is rounding up the result of the division, instead of rounding down in the `floor` division.

```
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b > 0);
        uint256 c = a / b;
        return c;
    }
```

Listing 3.8: div() in SafeMath

The current implementation of `SafeMath` does not support the `ceiling` division operation, namely `divCeil`. And the lack of `divCeil` may introduce elusive precision loss. Especially in an AMM-based DEX scenario where a user trades in one token for another, if there is a rounding issue, it is always preferable to calculate the trading amount in a way towards the liquidity pool to ensure the pool is balanced. Therefore, depending on specific cases, the calculation may often needs to replace the normal `floor` division with `divCeil`. In the following, we show an example `divCeil` implementation for `uint256`.

```
    function divCeil(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 quotient = div(a, b);
        uint256 remainder = a - quotient * b;
        if (remainder > 0) {
            return quotient + 1;
        } else {
            return quotient;
        }
```

```
    }
```

Listing 3.9: New divCeil () in SafeMath

Our analysis shows many functions in BlackHoleSwap requires `divCeil`. In the following, we explain our finding in three different scenarios:

Scenario I: Asset Trading

The first scenario is the trading behavior between the two supported assets: `DAI` and `USDC`. We use the `dai2usdcOut()` function as an example and its implementation is shown below.

```
383     function dai2usdcOut(uint256 max_input, uint256 output, uint256 deadline) public
            returns (uint256) {
384         require(block.timestamp <= deadline, "EXPIRED");
385         (uint256 a, uint256 b) = getDaiBalance();
386         (uint256 c, uint256 d) = getUSDCBalance();

388         uint256 input = getOutputPrice(output.mul(rate()), a, b, c, d);
389         securityCheck(input, output, a, b, c, d);
390         input = input.mul(1e18).div(fee);
391         require(input <= max_input, "SLIPPAGE_DETECTED");

393         calcFee(input, a, b, c, d);

395         doTransferIn(Dai, cDai, b, msg.sender, input);
396         doTransferOut(USDC, cUSDC, c.div(rate()), msg.sender, output);

398         emit Purchases(msg.sender, address(USDC), input, output);

400         return input;
401     }
```

Listing 3.10: dai2usdcOut()

This particular function provides the logic to trade `DAI` for `USDC`. The input amount of `DAI` is calculated as `input = input.mul(1e18).div(fee)` (line 390). However, the current implementation uses the default floor behavior of `div`. When it cannot be fully divided, the remainder portion of the division result is lost, meaning slightly less assets are transferred into the pool. A better approach is to replace the `div` calculation with `divCeil`, i.e., `input = input.mul(1e18).divCeil(fee)`. By doing so, we can guarantee the pool is always balanced.

A similar issue also occurs to the `usdc2daiOut()` function for the very same input amount calculation (line 410).

Scenario II: addLiquidity

The second scenario is the liquidity-supplying `addLiquidity()` behavior (see the code snippet below). The transfer-in amount of `USDC` is calculated in `usdc_amount = share.mul(usdc_reserve).div(`

`_totalSupply).div(rate())`. Similarly, potential rounding issue can be introduced to cause the loss on the pool size. A better approach is to calculate the amount as `usdc_amount = share.mul(usdc_reserve).divCeil(_totalSupply.mul(rate()))`.

```
466     function addLiquidity(uint256 share, uint256[2] memory tokens) public returns (
            uint256, uint256) {
467         require(share >= 1e15, 'INVALID_ARGUMENT'); // 1000 * rate()

469         collectComp();

471         if (_totalSupply > 0) {
472             (uint256 a, uint256 b) = getDaiBalance();
473             (uint256 c, uint256 d) = getUSDCBalance();

475             if(a < b) {
476                 uint256 dai_reserve = b.sub(a);
477                 uint256 usdc_reserve = c.sub(d);
478                 uint256 dai_amount = share.mul(dai_reserve).div(_totalSupply);
479                 uint256 usdc_amount = share.mul(usdc_reserve).div(_totalSupply);
480                 usdc_amount = usdc_amount.div(rate());
481                 require(dai_amount >= tokens[0] && usdc_amount <= tokens[1], "
                        SLIPPAGE_DETECTED");

483                 _mint(msg.sender, share);
484                 doTransferIn(USDC, cUSDC, d.div(rate()), msg.sender, usdc_amount);
485                 doTransferOut(Dai, cDai, a, msg.sender, dai_amount);

487                 emit AddLiquidity(msg.sender, dai_amount, usdc_amount);
488                 return (dai_amount, usdc_amount);

490             }
491             ...
492         }
493         ...
494     }
```

Listing 3.11: dai2usdcOut()

Similar rounding issues can also be caused at lines 494, 510 – 512, and 523 – 524 for various amount calculation of assets being transferred into the pool.

Scenario III: removeLiquidity

The third scenario is the liquidity-removing `removeLiquidity()` behavior (see the code snippet below). The transfer-in amount of `DAI` is calculated in `dai_amount = share.mul(dai_reserve).div(_totalSupply)` (line 546). Similarly, potential rounding issue can be introduced to cause the loss on the pool size. A better approach is to calculate the amount as `dai_amount = share.mul(dai_reserve).divCeil(_totalSupply)`.

```
    function removeLiquidity(uint256 share, uint256[2] memory tokens) public returns (
        uint256, uint256) {
        require(share > 0, 'INVALID_ARGUMENT');

        collectComp();

        (uint256 a, uint256 b) = getDaiBalance();
        (uint256 c, uint256 d) = getUSDCBalance();

        if(a < b) {
            uint256 dai_reserve = b.sub(a);
            uint256 usdc_reserve = c.sub(d);
            uint256 dai_amount = share.mul(dai_reserve).div(_totalSupply);
            uint256 usdc_amount = share.mul(usdc_reserve).div(_totalSupply);
            usdc_amount = usdc_amount.div(rate());
            require(dai_amount <= tokens[0] && usdc_amount >= tokens[1], "
                SLIPPAGE_DETECTED");

            _burn(msg.sender, share);
            doTransferIn(Dai, cDai, b, msg.sender, dai_amount);
            doTransferOut(USDC, cUSDC, c.div(rate()), msg.sender, usdc_amount);

            emit RemoveLiquidity(msg.sender, dai_amount, usdc_amount);
            return (dai_amount, usdc_amount);

        }
        ...
    }
```

Listing 3.12: dai2usdcOut()

Another similar issue can be found at lines 563 – 564 during the calculation of `usdc_amount` for being transferred into the pool.

Recommendation Revise the logic accordingly in the above three scenarios with `divCeil`.

3.5 Enriched Event Generation

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `blackholeswap`
- Category: Time and State [11]
- CWE subcategory: CWE-362 [7]

Description

Meaningful events are an important part in smart contract design as they can not only greatly expose the runtime dynamics of smart contracts, but also allow for better understanding about their behavior and facilitate off-chain analytics.

BlackHoleSwap defines three main events, i.e., `Purchases`, `AddLiquidity`, and `RemoveLiquidity`, to correspondingly record the results of asset trading, liquidity-supplying, and liquidity-removing behaviors. Each behavior may involve asset transfers insider or outside. Our analysis shows that each of these events can be enriched with additional information.

Using the `Purchases` event as an example, it is currently defined as `Purchases(address indexed buyer, address indexed buy_token, uint256 inputs, uint256 outputs)` with essential information such as current `buyer`, `buy_token`, `inputs`, and `outputs`. However, the valuable `sell_token` information is missing.

Also, the `AddLiquidity` event is defined as `AddLiquidity(address indexed provider, uint256 DAIAmount, uint256 USDCAmount)` with current `liquidity provider`, the provided `DAIAmount` and `USDCAmount`. It is important to note that the assets may move out of current pool. In other words, the direction of asset movement needs to be reflected in this event. Therefore, it is suggested to redefine the `AddLiquidity` event as follows: `AddLiquidity(address indexed provider, uint256 share, int256 DAIAmount, int256 USDCAmount)`. The difference is the type change from `uint256` to `int256` for both `DAIAmount` and `USDCAmount`. The signedness essentially reflects the asset movement direction. From the encoded direction, we can precisely pinpoint the particular asset reserve status inside the `addLiquidity` execution logic. In addition, we enclose the share information for better accounting and analytics.

The `RemoveLiquidity` event shares the same issue with `AddLiquidity` and can be readily revised in the same way as suggested with `AddLiquidity`.

Recommendation Revise the above three events by encoding more semantic information and better reflecting the protocol dynamics.

```
    event Purchases(address indexed buyer, address indexed sell_token, uint256
        sell_amount, address indexed buy_token, uint256 buy_amount);
    event AddLiquidity(address indexed provider, uint256 share, int256 DAIAmount, int256
         USDCAmount);
    event RemoveLiquidity(address indexed provider, uint256 share, int256 DAIAmount,
        int256 USDCAmount);
```

Listing 3.13: blackholeswapV1.sol (revised)

3.6 Safety Checks in Liquidity Addition and Removal

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `blackholeswap`
- Category: Time and State [11]
- CWE subcategory: CWE-362 [7]

Description

BlackHoleSwap acts as a trustless intermediary between liquidity providers and trading users. The liquidity providers `deposit` certain amount of `DAI/USDC` assets into the BlackHoleSwap as collateral and allow for traders to swap the assets. If one side of assets is insufficient, another side of assets is used as collateral in Compound to borrow the insufficient asset to meet the trading need.

There are two operations for liquidity provides, i.e., `addLiquidity` and `removeLiquidity`. As the names indicate, they allow for low-level routines to transfer liquidity assets into or out of BlackHoleSwap (see the code snippet below). Note that `addLiquidity` differentiates and processes four different scenarios while `removeLiquidity` handles three scenarios. In the following, we elaborate one typical scenario in `addLiquidity` to demonstrate the need of performing additional safety checks.

When adding additional liquidity into BlackHoleSwap, one particular scenario operates when the system currently deposits `USDC` as a collateral into Compound to borrow additional `DAI`s in order to meet rising trading demands.

```
    function addLiquidity(uint256 share, uint256[2] memory tokens) public returns (
        uint256, uint256) {
        require(share >= 1e15, 'INVALID_ARGUMENT'); // 1000 * rate()

        collectComp();

        if (_totalSupply > 0) {
            (uint256 a, uint256 b) = getDaiBalance();
            (uint256 c, uint256 d) = getUSDCBalance();

            if(a < b) {
                uint256 dai_reserve = b.sub(a);
                uint256 usdc_reserve = c.sub(d);
                uint256 dai_amount = share.mul(dai_reserve).div(_totalSupply);
                uint256 usdc_amount = share.mul(usdc_reserve).div(_totalSupply);
                usdc_amount = usdc_amount.div(rate());
                require(dai_amount >= tokens[0] && usdc_amount <= tokens[1], "
                    SLIPPAGE_DETECTED");

                _mint(msg.sender, share);
                doTransferIn(USDC, cUSDC, d.div(rate()), msg.sender, usdc_amount);
                doTransferOut(Dai, cDai, a, msg.sender, dai_amount);
```

```
                emit AddLiquidity(msg.sender, dai_amount, usdc_amount);
                return (dai_amount, usdc_amount);

            }
            ...
        }
        ...
    }
```

Listing 3.14: The addLiquidity() routine

The new addition of liquidity is allocated based on the requested share of the LP token's `_totalSupply`. And the share is equally applied for each asset, `DAI` and `USDC`. At first glance, if we assume the debt/collateral rate is safe right before the liquidity addition, the asset increase in terms of token amount should also meet the safe debt/collateral rate. However, the debt/collateral rate is measured based on the live market price and each asset may suffer from different fluctuations. As a result, the increase of one asset may not equally contribute to the calculation of debt/collateral rate.

This situation could be further exacerbated with the use of a flash loan attack. Note that a flash loan is only valid within one transaction and could fail if the borrower does not repay its debt before the end of the transaction borrowing the loan. However, with a flash loan, the actor may have a large volume of assets at the disposal and therefore could dramatically change the pool balance or asset distribution. In light of this, we strongly suggest to take a pre-cautious approach to apply necessary safety check when the provided liquidity is being changed.

Recommendation Apply necessary safety checks in both `addLiquidity` and `removeLiquidity` that could dynamically change the liquidity available for trading.

3.7 Recommended Reentrancy Protection

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `blackholeswap`
- Category: Concurrency [13]
- CWE subcategory: CWE-663 [9]

Description

The AMM-based exchange typically handles a variety of tokens and their governing contracts may be implemented in various forms and exhibit their noteworthy idiosyncrasies. For example, while some tokens may be fully-compliant with ERC20 standards, others may not. In addition, there exists deflationary tokens that may charge certain fee for every `transfer` or `transferFrom`. As a result, this may not meet the assumption behind low-level asset-transferring routines and unexpectedly introduce balance inconsistencies when comparing internal asset records with external balances maintained by the token contracts.

Other tokens may follow ERC20 standards but with additional customizations. For example, the ERC777 standard normalizes the ways to interact with a token contract while remaining backward compatible with ERC20. Among various features, it supports send/receive hooks to offer token holders more control over their tokens. Specifically, when `transfer` or `transferFrom` actions happen, the owner can be notified to make a judgment call so that she can control (or even reject) which token they send or receive by correspondingly registering a `tokensToSend` and `tokensReceived` hooks.

BlackHoleSwap has a number of entry points that are required to interact with external (untrusted) entities. While current prototype only supports a pair of well-defined tokens, `DAI` and `USDC`, we believe there is no reason to be concerned yet with various idiosyncrasies associated with tokens. However, BlackHoleSwap aims to advance the entire AMM frontline and needs to be broad in eventually accommodating a variety of tokens.

Accordingly, if many types of tokens are intended to be supported, possible reentrancy risks bring up the necessity to implement effective reentrancy prevention. In current prototype, the six main entries need be hardened for this purpose, i.e., `dai2usdcIn`, `usdc2daiIn`, `dai2usdcOut`, `usdc2daiOut`, `addLiquidity`, and `removeLiquidity`.

Recommendation Apply necessary reentrancy prevention by adding the following modifier to the above functions.

```

    bool internal locked;
    modifier noReentrancy() {
        require(!locked, "Reentrant call.");
        locked = true;
        _;
        locked = false;
    }
```

Listing 3.15: MarketContractProxy.sol

3.8 Possible Integer Overflow in *sqrt()*

- ID: PVE-008
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `SafeMath`
- Category: Numeric Errors [14]
- CWE subcategory: CWE-190 [3]

Description

The calculation of curve function `F()` requires finding the root of a quadratic equation and thus necessitates the familiar `sqrt()` function in order to calculate the integer square root of a given number. The `sqrt()` function, implemented in `SafeMath`, follows the `Babylonian` method for calculating the integer square root. Specifically, for a given x, we need to find out the largest integer z such that $z^2 <= x$.

```
71      function sqrt(int256 x) internal pure returns (int256) {
72          int256 z = ((add(x, 1)) / 2);
73          int256 y = x;
74          while (z < y)
75          {
76              y = z;
77              z = ((add((x / z), z)) / 2);
78          }
79          return y;
80      }
```

Listing 3.16: contracts/lib/SafeMath.sol

We show above current `sqrt()` implementation. The initial value of z to the iteration was given as $z = ((add(x, 1))/2)$, which results in an integer overflow when $x = max\_int256 = int256(2 ** 255 - 1)$. In other words, the overflow essentially sets z to zero, leading to a `division by zero` in the calculation of $z = ((add((x/z), z))/2)$ (line 77).

Note that this does not result in an incorrect return value from `sqrt()`, but does cause the function to revert unnecessarily when the above corner case occurs. Meanwhile, it is worth mentioning that if there is a `divide by zero`, the execution or the contract call will be thrown by executing the `INVALID` opcode, which by design consumes all of the gas in the initiating call. This is different from `REVERT` and has the undesirable result in causing unnecessary monetary loss.

To address this particular corner case, We suggest to change the initial value to $z = add(x/2, 1)$, making `sqrt()` well defined over its all possible inputs.

Recommendation Revise the above calculation to avoid the unnecessary integer overflow.

3.9 approve()/transferFrom() Race Condition

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Hakka`
- Category: Time and State [11]
- CWE subcategory: CWE-362 [7]

Description

`Hakka` is a standard ERC20 token that tokenizes the assets within a shared pool. In current implementation, there is a known race condition issue regarding `approve()` / `transferFrom()` [2]. Specifically, when a user intends to reduce the allowed spending amount previously approved from, say, 10 `HAKKA` to 1 `HAKKA`. The previously approved spender might race to transfer the amount you initially approved (the 10 `HAKKA`) and then additionally spend the new amount you just approved (1 `HAKKA`). This breaks the user's intention of restricting the spender to the new amount, **not** the sum of old amount and new amount. With the introduction of supporting `permit()`-based meta-transactions, a similar race condition also exists between `permit()/transferFrom()`.

In order to properly `approve` tokens, there also exists a known workaround: users can utilize the `increaseApproval` and `decreaseApproval` non-ERC20 functions on the token versus the traditional `approve` function.

```
    function approve(address spender, uint256 amount) external returns (bool) {
        allowance[msg.sender][spender] = amount;
        emit Approval(msg.sender, spender, amount);
        return true;
    }
```

Listing 3.17: Hakka.sol

Recommendation Add the suggested workaround functions `increaseApproval()/decreaseApproval()`. However, considering the difficulty and possible lean gains in exploiting the race condition, we also think it is reasonable to leave it as is.

3.10 Better Handling of Ownership Transfer

- ID: PVE-010
- Severity: Informational
- Likelihood: Low
- Impact: N/A
- Target: `Ownable.sol`
- Category: Security Features [10]
- CWE subcategory: CWE-282 [4]

Description

The `Ownable` smart contract implements a rather basic access control mechanism that allows a privileged account, i.e., `owner`, to be granted exclusive access to typically sensitive functions (e.g., the setting of certain risk parameters). Because of the `owner`-level access and the implications of these sensitive functions, the `owner` account is critical for the BlackHoleSwap security.

Within this contract, a specific function, i.e., `transferOwnership()`, allows for the ownership update. However, current implementation achieves its goal within a single transaction. This is reasonable under the assumption that the `newOwner` parameter is always correctly provided. However, in the unlikely situation, when an incorrect `newOwner` is provided, the contract ownership may be forever lost, which would be devastating for the entire system operation and maintenance.

As a common best practice, instead of achieving the ownership update within a single transaction, it is suggested to split the operation into two steps. The first step initiates the ownership update intention and the second step accepts and materializes the update. Both steps should be executed in two separate transactions. By doing so, it can greatly alleviate the concern of accidentally transferring the contract ownership to an uncontrolled address.

```
    function transferOwnership(address newOwner) public onlyOwner {
        require(newOwner != address(0), "invalid address");
        emit OwnershipTransferred(owner, newOwner);
        owner = newOwner;
    }
```

Listing 3.18: Ownable.sol

Recommendation Implement a two-step approach for ownership update (or transfer): `transferOwnership()` and `acceptOwnership()`.

```
    address public newOwner;

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address _newOwner) public onlyOwner {
```

```
        require(newOwner != _newOwner, "Ownable: new owner is the same as previous owner
            ");

        newOwner = _newOwner;
    }

    function acceptOwnership() public {
        require(msg.sender == newOwner);

        emit OwnershipTransferred(owner, newOwner);

        owner = newOwner;
        newOwner = 0x0;

    }
```

Listing 3.19: Revised Ownable.sol

3.11 Other Suggestions

Due to the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, we always suggest using fixed compiler version whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.6.10;` instead of `pragma solidity ^0.6.10;`.

Moreover, we strongly suggest not to use experimental Solidity features (e.g., `pragma experimental ABIEncoderV2`) or third-party unaudited libraries. If necessary, refactor current code base to only use stable features or trusted libraries.

Last but not least, 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 need to kick in at the very moment when the contracts are being deployed in the mainnet.

4 | Conclusion

In this audit, we thoroughly analyzed the BlackHoleSwap design and implementation. The proposed AMM-based DEX system presents a unique innovation and we are really impressed by the overall design and implementation. The current code base is well 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.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [17, 18, 19, 20, 22].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [23] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] HaleTom. Resolution on the EIP20 API Approve / TransferFrom multiple withdrawal attack. https://github.com/ethereum/EIPs/issues/738.

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

[4] MITRE. CWE-282: Improper Ownership Management. https://cwe.mitre.org/data/definitions/282.html.

[5] MITRE. CWE-284: Improper Access Control. https://cwe.mitre.org/data/definitions/284.html.

[6] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[7] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[8] MITRE. CWE-627: Dynamic Variable Evaluation. https://cwe.mitre.org/data/definitions/627.html.

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

[10] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[11] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

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

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

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

[17] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[18] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[19] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[20] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

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

[22] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[23] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.