SMART CONTRACT AUDIT REPORT

for

NODEEX HOLDINGS LIMITED

Prepared By: Shuxiao Wang

Hangzhou, China

September 6, 2020

Document Properties

Client	NODEEX HOLDINGS LIMITED
Title	Smart Contract Audit Report
Target	OneSwap
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Jeff Liu, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	September 6, 2020	Xuxian Jiang	Final Release
1.0-rc1	September 5, 2020	Xuxian Jiang	Release Candidate #1
0.4	September 4, 2020	Xuxian Jiang	Additional Findings #3
0.3	September 1, 2020	Xuxian Jiang	Additional Findings #2
0.2	August 29, 2020	Xuxian Jiang	Additional Findings #1
0.1	August 27, 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 **OneSwap** design document and related 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 branch of OneSwap 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 OneSwap

OneSwap is a fully decentralized exchange protocol on smart contract that uniquely supports both traditional order book (either market or limit orders) as well as automated market making (AMM). With permission-free token listing, users are able to establish liquidity pools without permission, and make markets through automated algorithms. It also has the plan to support liquidity mining and trade-driven mining simultaneously, providing both platform tokens and transaction fees as revenues. OneSwap pushes forward the current AMM-based DEX frontline and presents a valuable contribution to current DeFi ecosystem.

The basic information of OneSwap is as follows:

Table 1.1: Basic Information of OneSwap

Item	Description
Issuer	NODEEX HOLDINGS LIMITED
Website	https://www.oneswap.net/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 6, 2020

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- https://github.com/oneswap/oneswap_contract_ethereum (4194ac1)

1.2 About PeckShield

PeckShield Inc. [20] 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 [15]:

- 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) [14], 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 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 OneSwap 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. We also measured the gas consumption of key operations with comparison with the popular `UniswapV2`. The purpose here is to not only statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool, but also understand the performance in a realistic setting.

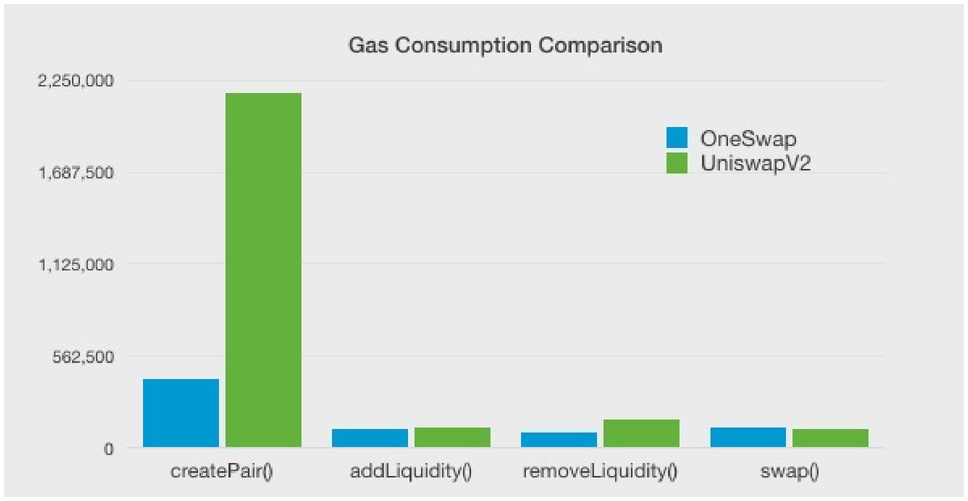

Figure 2.1: Gas Consumption Comparison Between `OneSwap` and `UniswapV2`

The performance comparison shows that `OneSwap` outperforms `UniswapV2` in almost all aspects despite the additional support of limit orders in a DEX setting. In particular, the adoption of a proxy-based approach (Section 3.9) greatly reduces the gas cost for the creation of a new pair. Also, a variety of optimization efforts, including the clever use of `immutable` members, the packed design of orders and other data structures, as well as the efficient communication between proxy and logic, eventually pay off even with the burden of integrated limit order support in `OneSwap`. Among all audited DeFi projects, `OneSwap` is exceptional and really stands out in their extreme quest and dedication to maximize gas optimization.

Severity	# of Findings	
Critical	0	
High	1	🟦
Medium	3	🟦🟦🟦🟦
Low	9	🟦🟦🟦🟦🟦🟦🟦🟦🟦
Informational	2	🟦🟦
Total	15	

Beside the performance measurement, we further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs. So far, we have 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 high-severity vulnerability, 3 medium-severity vulnerabilities, 9 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key OneSwap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Better Handling of Ownership Transfers	Business Logics	Fixed
PVE-002	High	Front-Running of Proposal Tallies	Time and State	Fixed
PVE-003	Low	Overlapped Time Windows Between Vote and Tally	Time and State	Fixed
PVE-004	Informational	Removal of Initial Nop Iterations in removeMainToken()	Coding Practices	Fixed
PVE-005	Medium	Incompatibility with Deflationary Tokens	Business Logics	Partially Fixed
PVE-006	Low	Tightened Sanity Checks in limitOrderWithETH()	Security Features	Fixed
PVE-007	Medium	Non-Payable removeLiquidityETH()	Coding Practices	Fixed
PVE-008	Low	Cached/Randomized ID For Unused OrderID Lookup	Coding Practices	Fixed
PVE-009	Medium	Gas-Efficient New Pair Deployment	Coding Practices	Fixed
PVE-010	Informational	Burnability of Assets Owned By Blacklisted Addresses	Business Logics	Fixed
PVE-011	Low	Accommodation of approve() Idiosyncrasies	Business Logics	Fixed
PVE-012	Low	Improved Handling of Corner Cases in SupervisedSend	Business Logics	Fixed
PVE-013	Low	Consistent Adherence of Checks-Effects-Interactions	Time and State	Fixed
PVE-014	Low	Improved Precision Calculation in Trading Fee Calculation	Coding Practice	Fixed
PVE-015	Low	Less Friction For Improved Buybacks and Order Matching	Coding Practice	Fixed

Please refer to Section 3 for details.

3 Detailed Results

3.1 Better Handling of Ownership Transfers

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `OneSwapBlackList`
- Category: Business Logics [12]
- CWE subcategory: CWE-841 [8]

Description

The `changeOwner()` function in the`OneSwapBlackList` contract allows the current owner of the contract to transfer her privilege to another address. However, in the ownership transfer implementation, the `newOwner` is directly stored into the storage, `_owner`, after only validating that the `newOwner` is a non-zero address (line 45).

```
26      function changeOwner(address newOwner) public override onlyOwner {
27          _setOwner(newOwner);
28      }

30      function addBlackLists(address[] calldata _evilUser) public override onlyOwner {
31          for (uint i = 0; i < _evilUser.length; i++) {
32              _isBlackListed[_evilUser[i]] = true;
33          }
34          emit AddedBlackLists(_evilUser);
35      }

37      function removeBlackLists(address[] calldata _clearedUser) public override onlyOwner
             {
38          for (uint i = 0; i < _clearedUser.length; i++) {
39              delete _isBlackListed[_clearedUser[i]];
40          }
41          emit RemovedBlackLists(_clearedUser);
42      }

44      function _setOwner(address newOwner) internal {
45          if (newOwner != address(0)) {
```

```
            _owner = newOwner;
            emit OwnerChanged(newOwner);
        }
    }
```

Listing 3.1: OneSwapBlackList.sol

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 owner may be forever lost, which might be devastating for OneSwap operation and maintenance.

As a common best practice, instead of achieving the owner update within a single transaction, it is suggested to split the operation into two steps. The first step initiates the owner update intent 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. In other words, this two-step procedure ensures that an owner public key cannot be nominated unless there is an entity that has the corresponding private key. This is explicitly designed to prevent unintentional errors in the owner transfer process.

Recommendation As suggested, the ownership transition can be better managed with a two-step approach, such as, using these two functions: `changeOwner()` and `acceptOwner()`. Specifically, the `changeOwner()` function keeps the new address in the storage, `_newOwner`, instead of modifying the `_owner` directly. The `acceptOwner()` function checks whether `_newOwner` is `msg.sender` to ensure that `_newOwner` signs the transaction and verifies herself as the new owner. Only after the successful verification, `_newOwner` would effectively become the `_owner`.

```
    function changeOwner(address newOwner) internal {
        require(newOwner != address(0), "Owner should not be 0 address");
        require(newOwner != _owner, "The current and new owner cannot be the same");
        require(newOwner != _newOwner, "Cannot set the candidate owner to the same
            address");
        _newOwner = newOwner;
    }

    function acceptOwner() public {
        require(msg.sender == _newOwner, "msg.sender and _newOwner must be the same");
        _owner = _newOwner;
        emit OwnershipTransferred(_owner, _newOwner);
    }
```

Listing 3.2: OneSwapBlackList.sol (revised)

Status The issue has been fixed by this commit: 49b5c8d0392e828b735445980e364d5ddc1c8542.

3.2 Front-Running of Proposal Tallies

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `OneSwapGov`
- Category: Time and State [10]
- CWE subcategory: CWE-362 [4]

Description

OneSwap defines a standard work-flow to submit, vote, and execute proposals that enact on the system-wide operations. There are four types of proposals, i.e., `_PROPOSAL_TYPE_FUNDS`, `_PROPOSAL_TYPE_PARAM`, `_PROPOSAL_TYPE_UPGRADE`, and `_PROPOSAL_TYPE_TEXT`. The `_PROPOSAL_TYPE_FUNDS` proposal allows for the allocation of certain `ones` assets to fund a particular project (or effort); the `_PROPOSAL_TYPE_PARAM` proposal enables dynamic configuration of system-wide protocol fee in BPS; the `_PROPOSAL_TYPE_UPGRADE` proposal allows for upgrade of the `OneSwap` DEX engine; the last type, i.e., `_PROPOSAL_TYPE_TEXT`, is currently a placeholder. The proposal falls in three different phases: `submit`, `vote`, and `tally`. The `tally` phase will immediately execute the proposal if passed.

Our analysis shows that the `tally()` function counts the user votes and is responsible for execute passed proposals. We notice the criteria of determining whether a proposal is passed is based on the balance sum of users who voted yes. And the balance is measured at the very moment when `tally()` occurs.

```
    // Count the votes, if the result is "Pass", transfer coins to the beneficiary
    function tally(uint64 proposalID, uint64 maxEntry) external override {
        Proposal memory proposal = proposals[proposalID];
        require(proposal.deadline != 0, "OneSwapGov: NO_PROPOSAL");
        // solhint-disable-next-line not-rely-on-time
        require(uint(proposal.deadline) <= block.timestamp, "OneSwapGov:
            DEADLINE_NOT_REACHED");
        require(maxEntry == _MAX_UINT64  (maxEntry > 0 && msg.sender == IOneSwapToken(
            ones).owner()),
            "OneSwapGov: INVALID_MAX_ENTRY");

        address currVoter = lastVoter[proposalID];
        require(currVoter != address(0), "OneSwapGov: NO_LAST_VOTER");
        uint yesCoinsSum = _yesCoins[proposalID];
        uint yesCoinsOld = yesCoinsSum;
        uint noCoinsSum = _noCoins[proposalID];
        uint noCoinsOld = noCoinsSum;

        for (uint64 i=0; i < maxEntry && currVoter != address(0); i++) {
            Vote memory v = votes[proposalID][currVoter];
            if(v.opinion == _YES) {
                yesCoinsSum += IERC20(ones).balanceOf(currVoter);
            }
            if(v.opinion == _NO) {
                noCoinsSum += IERC20(ones).balanceOf(currVoter);
            }
            delete votes[proposalID][currVoter];
            currVoter = v.prevVoter;
        }

        ...
    }
```

Listing 3.3: OneSwapGov.sol

As a result, if a malicious actor chooses to front-run the `tally()` transaction, with enough voting assets, the actor can largely control the `tally()` results. And flashloans can readily meet the need of enough voting assets for this front-running attack.

The fundamental reason while such attack is possible is due to the way how voting weights are calculated. Without locking up any asset to be committed for the votes, the proposal-based governance system carry less weight in the final results. Moreover, by only counting the voting weights when the `tally()` operation occurs and the `tally()` operation may not finish within a single transaction, it unnecessarily provides room for manipulation.

Recommendation Develop an effective counter-measure against the manipulation of `tally()` results.

Status The issue has been confirmed and fixed by proposing a new governance implementation in this commit: `49b5c8d0392e828b735445980e364d5ddc1c8542`. The new governance requires asset lockups to better serve the governance purposes.

3.3 Overlapped Time Windows Between Vote and Tally

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `OneSwapGov`
- Category: Time and State [10]
- CWE subcategory: CWE-362 [4]

Description

As described in Section 3.2, OneSwap defines a standard work-flow to submit, vote, and execute proposals that enact on the system-wide operations. A proposal falls in three different states, i.e., `submit`, `vote`, and `tally`. After a proposal is submitted, users can vote it within a pre-defined `_VOTE_PERIOD`.

This _VOTE_PERIOD is hard-coded constant of 3 days. After this voting period, the voted proposal can then be tallied to decide whether the proposal should be next executed or not.

Our analysis shows that the logic to enforce the voting period introduces a corner case that needs to be better handled. Specifically, as shown in the following code snippet, the proposal, once submitted, can be voted before the deadline, i.e., `deadline = uint32(block.timestamp + _VOTE_PERIOD )`. More precisely, any vote will be accepted if the following condition is satisfied: `require(uint( proposal.deadline)>= block.timestamp)` (line 108).

```
100      // Have never voted before, vote for the first time
101      function vote(uint64 id, uint8 opinion) external override {
102          uint balance = IERC20(ones).balanceOf(msg.sender);
103          require(balance > 0, "OneSwapGov: NO_ONES");

105          Proposal memory proposal = proposals[id];
106          require(proposal.deadline != 0, "OneSwapGov: NO_PROPOSAL");
107          // solhint-disable-next-line not-rely-on-time
108          require(uint(proposal.deadline) >= block.timestamp, "OneSwapGov:
                 DEADLINE_REACHED");

110          require(_YES<=opinion && opinion<=_NO, "OneSwapGov: INVALID_OPINION");
111          Vote memory v = votes[id][msg.sender];
112          require(v.opinion == 0, "OneSwapGov: ALREADY_VOTED");

114          v.prevVoter = lastVoter[id];
115          v.opinion = opinion;
116          votes[id][msg.sender] = v;

118          lastVoter[id] = msg.sender;

120          emit NewVote(id, msg.sender, opinion);
121      }
```

Listing 3.4: OneSwapGov.sol

For the `tally()` operation, it can be performed when the following timing is met, i.e., `require(uint (proposal.deadline)<= block.timestamp)`. Apparently, there is an overlap when `require(uint(proposal .deadline)= block.timestamp)`. If there is an ongoing voting transaction and the tally transaction included in the same block, whether the voting is counted based on the transaction ordering within this particular block. If the voting transaction is arranged earlier within the block, it will be counted. Otherwise, it will not! This certainly brings confusions and should be avoided.

Recommendation Ensure there is no overlap between the time windows for voting and tallying. We can either ensure `vote()` can only occur when `require(uint(proposal.deadline)> block.timestamp )` (so `tally()` can occur when `require(uint(proposal.deadline)<= block.timestamp)`) or `tally()` can only happen when `require(uint(proposal.deadline)< block.timestamp)` (so `vote()` can occur when `require(uint(proposal.deadline)>= block.timestamp)`), but not both.

```
    // Have never voted before, vote for the first time
    function vote(uint64 id, uint8 opinion) external override {
        uint balance = IERC20(ones).balanceOf(msg.sender);
        require(balance > 0, "OneSwapGov: NO_ONES");

        Proposal memory proposal = proposals[id];
        require(proposal.deadline != 0, "OneSwapGov: NO_PROPOSAL");
        // solhint-disable-next-line not-rely-on-time
        require(uint(proposal.deadline) > block.timestamp, "OneSwapGov: DEADLINE_REACHED
            ");

        require(_YES<=opinion && opinion<=_NO, "OneSwapGov: INVALID_OPINION");
        Vote memory v = votes[id][msg.sender];
        require(v.opinion == 0, "OneSwapGov: ALREADY_VOTED");

        v.prevVoter = lastVoter[id];
        v.opinion = opinion;
        votes[id][msg.sender] = v;

        lastVoter[id] = msg.sender;

        emit NewVote(id, msg.sender, opinion);
    }
```

Listing 3.5: OneSwapGov.sol (revised)

Status The issue has been confirmed and fixed by proposing a new governance implementation in this commit: `49b5c8d0392e828b735445980e364d5ddc1c8542`. The new governance implementation takes care of the above time overlaps between vote and tally.

3.4 Removal of Initial Nop Iterations in removeMainToken()

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `OneSwapBuyback`
- Category: Coding Practices [11]
- CWE subcategory: CWE-1041 [2]

Description

OneSwap has an interesting buy-back mechanism in place that can be used to purchase (and burn) the protocol tokens. It allows for flexible and dynamic configuration of so-called `main tokens` so that when the provided liquidity needs to withdraw, it only retains these `_mainTokens` (likely with greater liquidity/trading volume or more stable price).

When the OneSwap protocol is deployed, the contract's constructor will automatically place both `ones` and `WETH` as `main tokens`. These two will always be considered as `main tokens` and cannot be removed.

```
42     // remove token from main token list
43     function removeMainToken(address token) external override {
44         require(msg.sender == IOneSwapToken(ones).owner(), "OneSwapBuyback:
               NOT_ONES_OWNER");
45         require(token != ones, "OneSwapBuyback: REMOVE_ONES_FROM_MAIN");
46         require(token != weth, "OneSwapBuyback: REMOVE_WETH_FROM_MAIN");
47         if (_mainTokens[token]) {
48             _mainTokens[token] = false;
49             uint256 lastIdx = _mainTokenArr.length - 1;
50             for (uint256 i = 0; i < lastIdx; i++) {
51                 if (_mainTokenArr[i] == token) {
52                     _mainTokenArr[i] = _mainTokenArr[lastIdx];
53                     break;
54                 }
55             }
56             _mainTokenArr.pop();
57         }
58     }
```

Listing 3.6: OneSwapBuyback.sol

Therefore, the `removeMainToken()` routine (that is tasked to dynamically remove a given main token) can be slightly optimized to skip the checking of these two coins. And these two coins always occupy the first two slots in the internal `_mainTokens` array.

Recommendation Optimize the `removeMainToken()` logic as the first two are pre-occupied and cannot be removed as shown below (line 50).

```
42     // remove token from main token list
43     function removeMainToken(address token) external override {
44         require(msg.sender == IOneSwapToken(ones).owner(), "OneSwapBuyback:
               NOT_ONES_OWNER");
45         require(token != ones, "OneSwapBuyback: REMOVE_ONES_FROM_MAIN");
46         require(token != weth, "OneSwapBuyback: REMOVE_WETH_FROM_MAIN");
47         if (_mainTokens[token]) {
48             _mainTokens[token] = false;
49             uint256 lastIdx = _mainTokenArr.length - 1;
50             for (uint256 i = 2; i < lastIdx; i++) {
51                 if (_mainTokenArr[i] == token) {
52                     _mainTokenArr[i] = _mainTokenArr[lastIdx];
53                     break;
54                 }
55             }
56             _mainTokenArr.pop();
57         }
58     }
```

Listing 3.7: OneSwapBuyback.sol

Status The issue has been fixed by this commit: `49b5c8d0392e828b735445980e364d5ddc1c8542`.

3.5 Incompatibility with Deflationary Tokens

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `OneSwapRouter`
- Category: Business Logics [12]
- CWE subcategory: CWE-841 [8]

Description

In OneSwap, the `OneSwapRouter` contract is designed to be the main entry for interaction with trading users. In particular, one entry routine, i.e., `swapToken()`, accepts asset transfer-in and swaps it for another. Naturally, the contract implements a number of low-level helper routines to transfer assets into or out of OneSwap. 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 _swap(address input, uint amountIn, address[] memory path, address _to)
        internal virtual returns (uint[] memory amounts) {
        amounts = new uint[](path.length + 1);
        amounts[0] = amountIn;

        for (uint i = 0; i < path.length; i++) {
            (address to, bool isLastSwap) = i < path.length - 1 ? (path[i+1], false) : (
                _to, true);
            amounts[i + 1] = IOneSwapPair(path[i]).addMarketOrder(input, to, uint112(
                amounts[i]), isLastSwap);
            if (!isLastSwap) {
                (address stock, address money)= _getTokensFromPair(path[i]);
                input = (stock != input) ? stock : money;
            }
        }
    }

    function swapToken(address token, uint amountIn, uint amountOutMin, address[]
        calldata path,
        address to, uint deadline) external override ensure(deadline) returns (uint[]
            memory amounts) {

        require(path.length >= 1, "OneSwapRouter: INVALID_PATH");
        // ensure pair exist
        _getTokensFromPair(path[0]);
        _safeTransferFrom(token, msg.sender, path[0], amountIn);
        amounts = _swap(token, amountIn, path, to);
        require(amounts[path.length] >= amountOutMin, "OneSwapRouter:
            INSUFFICIENT_OUTPUT_AMOUNT");
    }
```

Listing 3.8: OneSwapRouter.sol

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every transfer or transferFrom. As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `swapToken()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of OneSwap and affects protocol-wide operation and maintenance.

A similar issue can also be found in `SupervisedSend`. One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `transfer` or `transferFrom` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `transfer` or `transferFrom` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into OneSwap for indexing. However, as a trustless intermediary, OneSwap may not be in the position to effectively regulate the entire process. Meanwhile, there exist certain assets that may exhibit control switches that can be dynamically exercised to convert into deflationary.

We need to point out that this issue can be traced back to the `Periphery` codebase of `UniswapV2`.

Recommendation To accommodate the support of possible deflationary tokens, it is better to check the balance before and after the `transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost.

Status The issue has been confirmed and accordingly fixed by measuring the balances right before the low-level asset transfer and right after the transfer. The difference is used to calculate the actually transferred asset amount.

3.6 Tightened Sanity Checks in limitOrderWithETH()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `OneSwapRouter`
- Category: Security Features [9]
- CWE subcategory: CWE-287 [3]

Description

As mentioned in Section 3.5, the `OneSwapRouter` contract is designed to be the main entry for interaction with trading users. Another specific entry routine, i.e., `limitOrderWithETH()`, allows for submitting a limit order involved with `ETH` trading. It conveniently wraps the deposited `ETHs` into `WETHs` in order to take advantage of the uniform, standardized trading interface of `OneSwapPair`.

```
183     function limitOrderWithETH(bool isBuy, address pair, uint prevKey, uint price,
            uint32 id,
184         uint stockAmount, uint deadline) external payable override ensure(deadline) {
185         (address stock, address money) = _getTokensFromPair(pair);
186         require(stock == weth  money == weth, "OneSwapRouter: PAIR_MISMATCH");
187         uint ethLeft;
188         {
189             (uint _stockAmount, uint _moneyAmount) = IOneSwapPair(pair).
                    calcStockAndMoney(uint64(stockAmount), uint32(price));
190             if (isBuy) {
191                 require(msg.value >= _moneyAmount, "OneSwapRouter:
                        INSUFFICIENT_INPUT_AMOUNT");
192                 ethLeft = msg.value - _moneyAmount;
193             }else{
194                 require(msg.value >= _stockAmount, "OneSwapRouter:
                        INSUFFICIENT_INPUT_AMOUNT");
195                 ethLeft = msg.value - _stockAmount;
196             }
197         }
198
199         IWETH(weth).deposit{value: msg.value - ethLeft}();
200         assert(IWETH(weth).transfer(pair, msg.value - ethLeft));
201         IOneSwapPair(pair).addLimitOrder(isBuy, msg.sender, uint64(stockAmount), uint32(
                price), id, uint72(prevKey));
202         if (ethLeft > 0) { _safeTransferETH(msg.sender, ethLeft); }
203     }
```

Listing 3.9: OneSwapRouter.sol

To elaborate the logic, we show above the code snippet of `limitOrderWithETH()`. The specific `ETH` wrapping requires that either `stock` or `money` needs to be the supported `WETH`: `require(stock == weth || money == weth, "OneSwapRouter: PAIR_MISMATCH")` (line 186). Apparently, this is necessary to prevent accidental deposits.

However, the condition can be further strengthened by requiring the pairing of `isBuy`. When `stock = WETH`, the submitted limited order has to be a `SELL` order, i.e., `isBuy = false`; When `money = WETH`, the submitted limited order has to be a `BUY` order, i.e., `isBuy = true`. The tightened sanity checks are very helpful to prevent accidental deposits of `ETH`s when `stock = WETH` and `isBuy = true`. Otherwise, the accidentally deposited assets will likely be `sync()`'ed into the pool reserve or `skim()`'ed by others.

Recommendation Tighten the above-mentioned sanity checks on `limitOrderWithETH()`.

```
    function limitOrderWithETH(bool isBuy, address pair, uint prevKey, uint price, uint32 id,
        uint stockAmount, uint deadline) external payable override ensure(deadline) {
        (address stock, address money) = _getTokensFromPair(pair);
        require((stock == weth && !isBuy)  (money == weth && isBuy), "OneSwapRouter: PAIR_MISMATCH");
        uint ethLeft;
        {
            (uint _stockAmount, uint _moneyAmount) = IOneSwapPair(pair).calcStockAndMoney(uint64(stockAmount), uint32(price));
            if (isBuy) {
                require(msg.value >= _moneyAmount, "OneSwapRouter: INSUFFICIENT_INPUT_AMOUNT");
                ethLeft = msg.value - _moneyAmount;
            }else{
                require(msg.value >= _stockAmount, "OneSwapRouter: INSUFFICIENT_INPUT_AMOUNT");
                ethLeft = msg.value - _stockAmount;
            }
        }

        IWETH(weth).deposit{value: msg.value - ethLeft}();
        assert(IWETH(weth).transfer(pair, msg.value - ethLeft));
        IOneSwapPair(pair).addLimitOrder(isBuy, msg.sender, uint64(stockAmount), uint32(price), id, uint72(prevKey));
        if (ethLeft > 0) { _safeTransferETH(msg.sender, ethLeft); }
    }
```

Listing 3.10: OneSwapRouter.sol

Status The issue has been confirmed and fixed by providing a native `ETH` support in OneSwap. In other words, it does not need the front-end wrapper of `WETH` in order to support `ETH`-related trading pairs. Note that the `UniswapV2` implementation still needs the `WETH` wrapper.

3.7 Non-Payable removeLiquidityETH()

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `OneSwapRouter`
- Category: Coding Practices [11]
- CWE subcategory: CWE-1041 [2]

Description

Within the `OneSwapRouter` contract, there is another entry routine, i.e., `removeLiquidityETH()`. This routine allows the pool's liquidity providers to remove liquidity from the pool. By transparently unwrapping `WETH`s into `ETH`, `removeLiquidityETH()` greatly facilitates user experience for native `ETH`s.

It is important to note that this routine is only supposed to accept the pool's liquidity tokens, not others including `ETH`s. Therefore, the current definition of `function removeLiquidityETH(address pair, uint liquidity, uint amountTokenMin, uint amountETHMin, address to, uint deadline)external override payable` may wrongfully allow to take users' accidental `ETH` deposit. To prevent that from happening, it is suggested to remove the keyword `payable` from the definition.

```
    function removeLiquidityETH(address pair, uint liquidity, uint amountTokenMin, uint
        amountETHMin,
        address to, uint deadline) external override ensure(deadline) payable returns (
            uint amountToken, uint amountETH) {

        address token;
        (address stock, address money) = _getTokensFromPair(pair);
        if (stock == weth) {
            token = money;
            (amountETH, amountToken) = _removeLiquidity(pair, liquidity, amountETHMin,
                amountTokenMin, address(this));
        } else if (money == weth) {
            token = stock;
            (amountToken, amountETH) = _removeLiquidity(pair, liquidity, amountTokenMin,
                amountETHMin, address(this));
        } else {
            require(false, "OneSwapRouter: PAIR_MISMATCH");
        }
        IWETH(weth).withdraw(amountETH);
        _safeTransferETH(to, amountETH);
        _safeTransfer(token, to, amountToken);
    }
```

Listing 3.11: OneSwapRouter.sol

Recommendation Remove the `payable` keyword from the `removeLiquidityETH()` definition.

Status The issue has been confirmed and fixed by providing a native ETH support in OneSwap. As mentioned in Section 3.6, it does not need the front-end wrapper of WETH for ETH-related trading pairs. By contrast, the UniswapV2 deployment still needs the WETH wrapper.

3.8 Cached/Randomized ID For Unused OrderID Lookup

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: OneSwapPair
- Category: Coding Practices [11]
- CWE subcategory: CWE-1041 [2]

Description

OneSwap has a built-in order-matching engine that maintains two singly-linked lists for pending buy and sell orders. Each order has a unique ID assigned. To ensure the order uniqueness, the routine _getUnusedOrderID() is responsible for ensuring the assigned (new) order always bears an unused order ID. (Note the uniqueness only needs to be maintained with buy orders or sell orders, not both.)

We notice that when the routine is tasked to find an unused order ID (by a given ID input of 0), it always starts from 1 to search for unused ID. Such ID assignment may not be optimal as it always starts from the same number. A better alternative may be to start from the last unused ID or even randomize the ID from the msg.sender or tx.origin. By doing so, it is likely to improve the hit rate of finding an unused ID.

```
    // Get an unused id to be used with new order
    function _getUnusedOrderID(bool isBuy, uint32 id) internal view returns (uint32) {
        if(id == 0) { // 0 is reserved
            id = 1;
        }
        for(uint32 i = 0; i < 100 && id <= _MAX_ID; i++) { //try 100 times
            if(!_hasOrder(isBuy, id)) {
                return id;
            }
            id++;
        }
        require(false, "OneSwap: CANNOT_FIND_VALID_ID");
        return 0;
    }
```

Listing 3.12: OneSwapRouter.sol

Recommendation For a new unused ID assignment, start the ID search from the last unused ID or a randomized ID.

Status The issue has been fixed by randomizing the starting ID for assignment. It basically implements a pseudo-random number generator: `id = uint32(uint(blockhash(block.number-1))^uint (tx.origin))& _MAX_ID`. Note that this pseudo-random number generator may not be sufficiently secure, but it suffices for the purpose of order ID assignment.

3.9 Gas-Efficient New Pair Deployment

- ID: PVE-009
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `OneSwapFactory`
- Category: Coding Practices [11]
- CWE subcategory: CWE-1041 [2]

Description

OneSwap acts as a trustless intermediary between liquidity providers and trading users. The liquidity providers `deposit` certain amount of `stock` and `money` assets into the OneSwap pool and in return get the tokenized pool share of current reserves. Later on, the liquidity providers can `withdraw` their own share by returning the pool tokens back to the pool. With assets in the pool, users can submit `swap` or `limit` orders and the trading price is determined according to the current order book and/or AMM price curve.

When the pool does not exist, the first liquidity provider's `addLiquidity()` operation will trigger the creation of the pool (via the `createPair()` function). As the name indicates, `createPair()` performs necessary sanity checks and then instantiates the pool contract creation (line 60): `OneSwapPair oneswap = new OneSwapPair(weth, stock, money, isOnlySwap, uint64(uint(10)**dec), priceMul, priceDiv )`.

```
60      function createPair(address stock, address money, bool isOnlySwap) external override
            returns (address pair) {
61          require(stock != money, "OneSwapFactory: IDENTICAL_ADDRESSES");
62          require(stock != address(0) && money != address(0), "OneSwapFactory:
                ZERO_ADDRESS");
63          uint moneyDec = uint(IERC20(money).decimals());
64          uint stockDec = uint(IERC20(stock).decimals());
65          require(23 >= stockDec && stockDec >= 0, "OneSwapFactory:
                STOCK_DECIMALS_NOT_SUPPORTED");
66          uint dec = 0;
67          if(stockDec >= 4) {
68              dec = stockDec - 4; // now 19 >= dec && dec >= 0
69          }
70          // 10**19 = 10000000000000000000
71          //  1<<64 = 18446744073709551616
72          uint64 priceMul = 1;
```

```
        uint64 priceDiv = 1;
        bool differenceTooLarge = false;
        if(moneyDec > stockDec) {
            if(moneyDec > stockDec + 19) {
                differenceTooLarge = true;
            } else {
                priceMul = uint64(uint(10)**(moneyDec - stockDec));
            }
        }
        if(stockDec > moneyDec) {
            if(stockDec > moneyDec + 19) {
                differenceTooLarge = true;
            } else {
                priceDiv = uint64(uint(10)**(stockDec - moneyDec));
            }
        }
        require(!differenceTooLarge, "OneSwapFactory: DECIMALS_DIFF_TOO_LARGE");
        bytes32 salt = keccak256(abi.encodePacked(stock, money, isOnlySwap));
        require(_tokensToPair[salt] == address(0), "OneSwapFactory: PAIR_EXISTS");
        OneSwapPair oneswap = new OneSwapPair{salt: salt}(weth, stock, money, isOnlySwap
            , uint64(uint(10)**dec), priceMul, priceDiv);

        pair = address(oneswap);
        allPairs.push(pair);
        _tokensToPair[salt] = pair;
        _pairWithToken[pair] = TokensInPair(stock, money);
        emit PairCreated(pair, stock, money, isOnlySwap);
    }
```

Listing 3.13: OneSwapFactory.sol

The pair contract is a complicated one and its instantiation inevitably consumes significant amount of gas. Such gas-consuming pool contract deployment would discourage liquidity providers' engagement. An alternative would be to explore a proxy-based approach by implementing the pool contract as a logic one. By doing so, we only need to deploy a minimal proxy for each pair, hence lowering the entry barrier for liquidity providers, especially for the creation of trading pools. Recall that in order to prevent the first liquidity provider from monopolizing the liquidity pool, the provider has been penalized by forcibly burning the very first `_MINIMUM_LIQUIDITY = 10 ** 3` pool shares. It is just not justifiable to further penalize early liquidity providers who introduce the trading pools into the OneSwap ecosystem!

Recommendation Explore the proxy-based approach of deploying pool contracts to lower the barrier for early participation.

Status The issue has been confirmed. The team has seriously taken the suggested approach by implementing a proxy-based architecture in this commit: `d76898b603aed60a776fc0ac529b199e1a6c8c9e`. The benefit in reduced gas consumption is evident. In the following, we show the comparison of key

Table 3.1: Gas Consumption Comparison Between `OneSwap` And `UniswapV2`

Operation	OneSwap	UniswapV2	Note
createPair()	$419,493$	$2,174,541$	a 90% reduction from $> 5M$ gas to current $< 500K$
addLiquidity()	$116,409$	$123,702$	add new liquidity into the pool
removeLiquidity()	$97,837$	$175,966$	remove liquidity from the pool
swap()	$121,790$	$117,503$	swap one token to another against the pool

operations between `OneSwap` And `UniswapV2`.

3.10 Burnability of Assets Owned By Blacklisted Addresses

- ID: PVE-010
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `OneSwapToken`
- Category: Business Logics [12]
- CWE subcategory: CWE-754 [7]

Description

`OneSwapToken` defines the protocol token used in OneSwap. It supports the capability of blacklisting certain accounts. Note the `OneSwapTokens` owned by a blacklisted address are prevented from transferring to another account.

```
    function _transfer(address sender, address recipient, uint256 amount) internal
        virtual {
        require(sender != address(0), "OneSwapToken: TRANSFER_FROM_THE_ZERO_ADDRESS");
        require(recipient != address(0), "OneSwapToken: TRANSFER_TO_THE_ZERO_ADDRESS");

        _beforeTokenTransfer(sender, recipient, amount);

        _balances[sender] = _balances[sender].sub(amount, "OneSwapToken:
            TRANSFER_AMOUNT_EXCEEDS_BALANCE");
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, amount);
    }

    function _burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "OneSwapToken: BURN_FROM_THE_ZERO_ADDRESS");

        _balances[account] = _balances[account].sub(amount, "OneSwapToken:
            BURN_AMOUNT_EXCEEDS_BALANCE");
        _totalSupply = _totalSupply.sub(amount);
        emit Transfer(account, address(0), amount);
    }
    function _approve(address owner, address spender, uint256 amount) internal virtual {
        require(owner != address(0), "OneSwapToken: APPROVE_FROM_THE_ZERO_ADDRESS");
        require(spender != address(0), "OneSwapToken: APPROVE_TO_THE_ZERO_ADDRESS");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }

    function _beforeTokenTransfer(address from, address to, uint256 ) internal virtual
        view {
        require(!isBlackListed(from), "OneSwapToken: FROM_IS_BLACKLISTED_BY_TOKEN_OWNER"
            );
        require(!isBlackListed(to), "OneSwapToken: TO_IS_BLACKLISTED_BY_TOKEN_OWNER");
    }
```

Listing 3.14: OneSwapToken.sol

The blocking logic is implemented by invoking a call to `_beforeTokenTransfer()` that in essence answers whether any of the involved parties is blacklisted. If yes, the transfer is simply reverted. Meanwhile, we notice that a blacklisted account can still burn the owned assets. Ethically, we believe it is more appropriate to freeze the blacklisted account, including the `burn()` attempt by the blacklisted account.

Recommendation Add the support of preventing a blacklisted account from burning owned tokens. And also block a blacklisted account from spending if there is still pending allowance.

```
    function _burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "OneSwapToken: BURN_FROM_THE_ZERO_ADDRESS");
        require(!isBlackListed(account), "OneSwapToken:
            BURN_FROM_THE_BLACKLISTED_ADDRESS");

        _balances[account] = _balances[account].sub(amount, "OneSwapToken:
            BURN_AMOUNT_EXCEEDS_BALANCE");
        _totalSupply = _totalSupply.sub(amount);
        emit Transfer(account, address(0), amount);
    }
```

Listing 3.15: OneSwapToken.sol

Status The issue has been fixed by this commit: `49b5c8d0392e828b735445980e364d5ddc1c8542`.

3.11 Accommodation of approve() Idiosyncrasies

- ID: PVE-011
- Severity: Low
- Likelihood: medium
- Impact: Low
- Target: `OneSwapBuyback`
- Category: Business Logics [12]
- CWE subcategory: N/A

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.16: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. An example is shown below. It is in the `OneSwapBuyback` contract that is designed to swap certain tokens to the protocol token. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

```
    function _swapForOnesAndBurn(address pair) private {
        (address a, address b) = IOneSwapFactory(factory).getTokensFromPair(pair);
        require(a != address(0) && b != address(0), "OneSwapBuyback: INVALID_PAIR");
        require(a == ones  b == ones, "OneSwapBuyback: ONES_NOT_IN_PAIR");

        address token = (a == ones) ? b : a;
        require(_mainTokens[token], "OneSwapBuyback: MAIN_TOKEN_NOT_IN_PAIR");
        uint256 tokenAmt = IERC20(token).balanceOf(address(this));
        require(tokenAmt > 0, "OneSwapBuyback: NO_MAIN_TOKENS");

        address[] memory path = new address[](1);
        path[0] = pair;

        // token -> ones
        IERC20(token).approve(router, tokenAmt);
        IOneSwapRouter(router).swapToken(
            token, tokenAmt, 0, path, address(this), _MAX_UINT256);
    }
```

Listing 3.17: OneSwapBuyback.sol

Recommendation Accommodate the above-mentioned idiosyncrasy of `approve()`.

Status The issue has been fixed by this commit: `49b5c8d0392e828b735445980e364d5ddc1c8542`.

3.12 Improved Handling of Corner Cases in SupervisedSend

- ID: PVE-012
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SupervisedSend`
- Category: Business Logics [12]
- CWE subcategory: N/A

Description

OneSwap provides a number of unique features and `SupervisedSend` is one of them. `SupervisedSend` allows for effective lock-up of assets and the locked assets can be released after the lock-up expires.

The `SupervisedSend` contract has exposed a number of functions, including `supervisedSend()` and `supervisedUnlockSend()`. The first function properly locks up the assets with a specified expiry time and the second one allows for unlocking the assets after the lockup is expired. The lockup time is facilitated with two modifiers, i.e., `beforeUnlockTime` and `afterUnlockTime`. We show these two modifiers below.

```
    modifier afterUnlockTime(uint32 unlockTime) {
        require(uint(unlockTime) * 3600 < block.timestamp, "SupervisedSend:
            NOT_ARRIVING_UNLOCKTIME_YET");
```

```
22          _;
23      }
24
25      modifier beforeUnlockTime(uint32 unlockTime) {
26          require(uint(unlockTime) * 3600 > block.timestamp, "SupervisedSend:
                ALREADY_UNLOCKED");
27          _;
28      }
```

Listing 3.18: SupervisedSend.sol

Apparently, the `beforeUnlockTime` modifier ensures the assets are currently locked (`require(uint(unlockTime)* 3600 < block.timestamp` – line 21) and the `afterUnlockTime` modifier guarantees that the lockup period is over (`require(uint(unlockTime)* 3600 > block.timestamp` – line 26). It is interesting to note an un-handled corner case when `uint(unlockTime)* 3600 == block.timestamp`.

Another corner issue is also identified in the `_tryDealInPool()` routine of the `OneSwapPair` contract (line 1018).

Recommendation Address the missed corner cases without any omission.

Status The issue has been fixed by including the `=` case in the `afterUnlockTime` modifier. The corner case in `_tryDealInPool()` was fixed by this commit: `4194ac1a55934cd573bd93987111eaa8f70676fe`.

```
20      modifier afterUnlockTime(uint32 unlockTime) {
21          require(uint(unlockTime) * 3600 <= block.timestamp, "SupervisedSend:
                NOT_ARRIVING_UNLOCKTIME_YET");
22          _;
23      }
24
25      modifier beforeUnlockTime(uint32 unlockTime) {
26          require(uint(unlockTime) * 3600 > block.timestamp, "SupervisedSend:
                ALREADY_UNLOCKED");
27          _;
28      }
```

Listing 3.19: SupervisedSend.sol (revised)

3.13 Consistent Adherence of Checks-Effects-Interactions

- ID: PVE-013
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SupervisedSend`
- Category: Time and State [13]
- CWE subcategory: CWE-663 [6]

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` [23] exploit, and the recent `Uniswap/Lendf.Me` hack [21].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `SupervisedSend` as an example, the `supervisedSend()` function (see the code snippet below) is provided to externally call a token contract to transfer assets for lock-up. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 36) starts before effecting the update on the internal state (line 37), hence violating the principle. In this particular case, if the external contract has some hidden logic that may be capable of launching `re-entrancy` via the very same `supervisedSend()` function. Meanwhile, we should emphasize that the `ones` tokens implement rather standard ERC20 interfaces and its token contract is not vulnerable or exploitable for `re-entrancy`.

```
30    function supervisedSend(address to, address supervisor, uint112 reward, uint112
          amount, address token, uint32 unlockTime, uint256 serialNumber) public override
          {
31        bytes32 key = _getSupervisedSendKey(msg.sender, to, supervisor, token,
              unlockTime);
32        supervisedSendInfo memory info = supervisedSendInfos[key][serialNumber];
33        require(amount > reward, "SupervisedSend: TOO_MUCH_REWARDS");
34        // prevent duplicated send
35        require(info.amount == 0 && info.reward == 0, "SupervisedSend:
              INFO_ALREADY_EXISTS");
36        _safeTransferToMe(token, msg.sender, uint(amount).add(uint(reward)));
37        supervisedSendInfos[key][serialNumber]= supervisedSendInfo(amount, reward);
38        emit SupervisedSend(msg.sender, to, supervisor, token, amount, reward,
              unlockTime);
39    }
```

Listing 3.20: SupervisedSend.sol

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

```
    function supervisedSend(address to, address supervisor, uint112 reward, uint112
        amount, address token, uint32 unlockTime, uint256 serialNumber) public override
        {
        bytes32 key = _getSupervisedSendKey(msg.sender, to, supervisor, token,
            unlockTime);
        supervisedSendInfo memory info = supervisedSendInfos[key][serialNumber];
        require(amount > reward, "SupervisedSend: TOO_MUCH_REWARDS");
        // prevent duplicated send
        require(info.amount == 0 && info.reward == 0, "SupervisedSend:
            INFO_ALREADY_EXISTS");
        supervisedSendInfos[key][serialNumber]= supervisedSendInfo(amount, reward);
        _safeTransferToMe(token, msg.sender, uint(amount).add(uint(reward)));
        emit SupervisedSend(msg.sender, to, supervisor, token, amount, reward,
            unlockTime);
    }
```

Listing 3.21: SupervisedSend.sol (revised)

Status This issue has been fixed by following the `checks-effects-interactions` best practice.

3.14 Improved Precision Calculation in Trading Fee Calculation

- ID: PVE-014
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `OneSwapPair`
- Category: Coding Practices [11]
- CWE subcategory: CWE-627 [5]

Description

`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. In this section, we examine one possible precision loss source that stems 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.

During the analysis of an internal function, i.e, `_dealWithPoolAndCollectFee()`, that makes a deal with the pool and then collects necessary fee, we notice the fee calculation results in (small) precision loss. For elaboration, we show the related code snippet below.

```
1066    // make real deal with the pool and then collect fee, which will be added to AMM pool
1067    function _dealWithPoolAndCollectFee(Context memory ctx, bool isBuy) internal returns (uint) {
1068        (uint outpoolTokenReserve, uint inpoolTokenReserve, uint otherToTaker) = (
1069              ctx.reserveMoney, ctx.reserveStock, ctx.dealMoneyInBook);
1070        if(isBuy) {
1071            (outpoolTokenReserve, inpoolTokenReserve, otherToTaker) = (
1072                ctx.reserveStock, ctx.reserveMoney, ctx.dealStockInBook);
1073        }

1075        // all these 4 varialbes are less than 112 bits
1076        // outAmount is sure to less than outpoolTokenReserve (which is ctx.reserveStock or ctx.reserveMoney)
1077        uint outAmount = (outpoolTokenReserve*ctx.amountIntoPool)/(inpoolTokenReserve+ctx.amountIntoPool);
1078        if(ctx.amountIntoPool > 0) {
1079            _emitDealWithPool(uint112(ctx.amountIntoPool), uint112(outAmount), isBuy);
1080        }
1081        uint32 feeBPS = IOneSwapFactory(ctx.factory).feeBPS();
1082        // the token amount that should go to the taker,
1083        // for buy-order, it's stock amount; for sell-order, it's money amount
1084        uint amountToTaker = outAmount + otherToTaker;
1085        require(amountToTaker < uint(1<<112), "OneSwap: AMOUNT_TOO_LARGE");
1086        uint fee = amountToTaker * feeBPS / 10000;
1087        amountToTaker -= fee;

1089        if(isBuy) {
1090            ctx.reserveMoney = ctx.reserveMoney + ctx.amountIntoPool;
1091            ctx.reserveStock = ctx.reserveStock - outAmount + fee;
1092        } else {
1093            ctx.reserveMoney = ctx.reserveMoney - outAmount + fee;
1094            ctx.reserveStock = ctx.reserveStock + ctx.amountIntoPool;
1095        }

1097        address token = ctx.moneyToken;
1098        if(isBuy) {
1099            token = ctx.stockToken;
1100        }
1101        _safeTransfer(token, ctx.order.sender, amountToTaker, ctx.ones);
1102        return amountToTaker;
1103    }
```

Listing 3.22: OneSwapPair()

The fee calculation is performed via `fee = amountToTaker * feeBPS / 10000` (line 1086). Apparently, it is a standard `floor()` operation that rounds down the calculation result. Note that 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 protect the liquidity providers' interest. Therefore, depending on specific cases, the calculation may often needs to replace the normal `floor` division with `ceiling` division. In other words, the fee calculation is better revised as `fee = (amountToTaker * feeBPS + 9999)/ 10000`, a `ceiling` division.

Recommendation Revise the logic accordingly to round-up the fee calculation.

Status The issue has been confirmed and fixed by taking the suggested round-up approach for the fee calculation.

3.15 Less Friction For Improved Buybacks and Order Matching

- ID: PVE-015
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OneSwapBuyback`, `OneSwapPair`
- Category: Coding Practices [11]
- CWE subcategory: N/A

Description

OneSwap has a number of components that not only depend on each other, but also interact with external DeFi protocols. Because of that, it is often necessary to introduce as little friction as possible to avoid sudden disruption of an ongoing transaction. Note that the disruption can be caused by imposed requirements on the related execution paths. Certainly, essential requirements need to be satisfied while others need to gauge specific application situations or logics to avoid unnecessary or sudden `revert`.

In the following, we show a specific case in the `OneSwapBuyback` contract. The specific function is `_removeLiquidity()`. As the name indicates, it allows previously provided liquidity to be removed from the pool. For convenience, it further supports batch-processing: given a list of pairs (and their associated liquidity pools), it iterates each one and removes the provided liquidity. However, it also requires `require(amt > 0, "OneSwapBuyback: NO_LIQUIDITY")` (line 81). This requirement will unnecessarily revert the ongoing transaction even if we can simply skip it during the batch processing.

```
69      // remove Buyback's liquidity from all pairs
70      // swap got minor tokens for main tokens if possible
71      function removeLiquidity(address[] calldata pairs) external override {
72          for (uint256 i = 0; i < pairs.length; i++) {
73              _removeLiquidity(pairs[i]);
74          }
75      }
76      function _removeLiquidity(address pair) private {
77          (address a, address b) = IOneSwapFactory(factory).getTokensFromPair(pair);
```

```
        require(a != address(0)  b != address(0), "OneSwapBuyback: INVALID_PAIR");

        uint256 amt = IERC20(pair).balanceOf(address(this));
        require(amt > 0, "OneSwapBuyback: NO_LIQUIDITY");

        IERC20(pair).approve(router, 0);
        IERC20(pair).approve(router, amt);
        IOneSwapRouter(router).removeLiquidity(
            pair, amt, 0, 0, address(this), _MAX_UINT256);

        // minor -> main
        bool aIsMain = _mainTokens[a];
        bool bIsMain = _mainTokens[b];
        if ((aIsMain && !bIsMain)  (!aIsMain && bIsMain)) {
            _swapForMainToken(pair);
        }
    }
```

Listing 3.23: OneSwapBuyback.sol

Another similar issue can also be found in the `_intopoolAmountTillPrice()` routine in the `OneSwapPair` contract.

Recommendation Introduce as little friction as possible by revising the `_removeLiquidity()` routine accordingly.

```
    // remove Buyback's liquidity from all pairs
    // swap got minor tokens for main tokens if possible
    function removeLiquidity(address[] calldata pairs) external override {
        for (uint256 i = 0; i < pairs.length; i++) {
            _removeLiquidity(pairs[i]);
        }
    }
    function _removeLiquidity(address pair) private {
        (address a, address b) = IOneSwapFactory(factory).getTokensFromPair(pair);
        require(a != address(0)  b != address(0), "OneSwapBuyback: INVALID_PAIR");

        uint256 amt = IERC20(pair).balanceOf(address(this));
        if (amt == 0) { return };

        IERC20(pair).approve(router, 0);
        IERC20(pair).approve(router, amt);
        IOneSwapRouter(router).removeLiquidity(
            pair, amt, 0, 0, address(this), _MAX_UINT256);

        // minor -> main
        bool aIsMain = _mainTokens[a];
        bool bIsMain = _mainTokens[b];
        if ((aIsMain && !bIsMain)  (!aIsMain && bIsMain)) {
            _swapForMainToken(pair);
        }
    }
```

Listing 3.24: OneSwapBuyback.sol

Status The issue has been fixed by replacing non-essential `require()` with corresponding `if` conditions.

3.16 Other Suggestions

OneSwap merges the DEX support of traditional order book and automated market making. While it greatly pushes forward the DEX frontline, it also naturally inherits from well-known front-running or back-running issues plagued with current DEXs. For example, a large trade may be sandwiched by preceding addition into liquidity pool (via `mint()`) and tailgating removal of the same amount of liquidity (via `burn()`). Such sandwiching unfortunately causes a loss to other liquidity providers. Also, a large burn of the protocol token (via the built-in `buyback` mechanism) could be similarly sandwiched by preceding buys for increased token values. Similarly, a market order could be intentionally traded for a higher price if a malicious actor intentionally increases it by trading an earlier competing order. However, we need to acknowledge that these are largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Next, because the Solidity language is still maturing and it is common for new compiler versions to include changes that might bring unexpected compatibility or inter-version consistencies, it is always suggested to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.6.6;` instead of `pragma solidity >=0.6.6 or ^0.6.6;`.

Moreover, we strongly suggest not to use experimental Solidity features 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 mainnet.

4 | Conclusion

In this audit, we thoroughly analyzed the OneSwap design and implementation. The system presents a unique offering in current DEX ecosystem with the support of both traditional order book and AMMs. We are truly impressed by the design and implementation, especially the dedication to maximized gas optimization. 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 [16, 17, 18, 19, 22].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [24] 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] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

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

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

[7] MITRE. CWE-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre.org/data/definitions/754.html.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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