SMART CONTRACT AUDIT REPORT

for

SatoshiIncognito DaoSwap

Prepared By: <u>Xiaomi Huang</u>

PeckShield
August 10, 2022

Document Properties

Client	SatoshiIncognito
Title	Smart Contract Audit Report
Target	DaoSwap
Version	1.0-rc
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc1	August 10, 2022	Luck Hu	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 | Introduction

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

1.1 About DaoSwap

`SatoshiIncognito` DaoSwap is a professional trading platform on `BSC`, which is initiated and managed by the `DaoSwap Foundation`. The protocol supports users to freely create trading pairs and earn transaction fees to achieve decentralized token exchange and automatic market making. Users can earn swap transaction mining by trading in `DaoSwap`, and receive airdrop rewards by referring users to trade in `DaoSwap`. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The `DaoSwap` Protocol

Item	Description
Issuer	SatoshiIncognito
Website	http://daoswap.financial/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 10, 2022

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

- https://github.com/SatoshiIncognito/DaoSwap.git (4833115)

And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/SatoshiIncognito/DaoSwap.git (TBD)

1.2   About PeckShield

PeckShield Inc. [9] 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

	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Impact (vertical axis) — Likelihood (horizontal axis)

1.3   Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- 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:

- <u>Basic Coding Bugs</u>: 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.

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

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

- <u>Additional Recommendations</u>: 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) [7], 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 security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

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

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

PeckShield Audit Report #: 2022-306

2 | Findings

2.1 Summary

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

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	4	■■■■
Informational	0	
Total	6	

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 that 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 and 4 low-severity vulnerabilities.

Table 2.1: Key DaoSwap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Inconsistent Fee Rate Used in DaoSwap	Business Logic	Fixed
PVE-002	Low	Improved Sanity Checks For System Parameters	Coding Practices	Fixed
PVE-003	Low	Incompatibility With Deflationary Tokens	Business Logic	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-005	Low	Potential Arbitrage Swaps To Drain All Rewards	Business Logic	Confirmed
PVE-006	Low	Proper Management For The Invitation Rewards	Coding Practices	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Inconsistent Fee Rate Used in DaoSwap

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: DAOLibrary, DAOPair
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In SatoshiIncognito DaoSwap, there is a DAOLibrary contract which provides a pair of interfaces (i.e., getAmountIn()/getAmountOut()) to facilitate users trading. While reviewing the swap fee rate used in the two interfaces, we notice it is inconsistent with the one used in the swap() and the inconsistency needs to be resolved.

To elaborate, we show below the code snippet of the getAmountIn()/getAmountOut()/swap() routines. As the name indicates, the getAmountIn() routine accepts an input amount of an asset and pair reserves, and returns the maximum output amount of the other asset. The getAmountOut() counterpart accepts an output amount of an asset and pair reserves, and returns a required input amount of the other asset. The fee rate used in these two routines is 3/1000 (lines 47 and 58). However, in the swap() routine which implements the trading, the fee rate used is 25/10000 (lines 198 and 199). Due to the inconsistent fee rates, the user may not get the expected amount of asset from the swap or the getAmountIn()/getAmountOut() routines may not return the correct token amount to the user.

```
43      // given an input amount of an asset and pair reserves, returns the maximum output
            amount of the other asset
44      function getAmountOut(uint amountIn, uint reserveIn, uint reserveOut) internal pure
            returns (uint amountOut) {
45          require(amountIn > 0, 'DAOLibrary: INSUFFICIENT_INPUT_AMOUNT');
46          require(reserveIn > 0 && reserveOut > 0, 'DAOLibrary: INSUFFICIENT_LIQUIDITY');
47          uint amountInWithFee = amountIn.mul(9970);
48          uint numerator = amountInWithFee.mul(reserveOut);
49          uint denominator = reserveIn.mul(10000).add(amountInWithFee);
```

PeckShield Audit Report #: 2022-306

```
50          amountOut = numerator / denominator;
51      }
52
53      // given an output amount of an asset and pair reserves, returns a required input
            amount of the other asset
54      function getAmountIn(uint amountOut, uint reserveIn, uint reserveOut) internal pure
            returns (uint amountIn) {
55          require(amountOut > 0, 'DAOLibrary: INSUFFICIENT_OUTPUT_AMOUNT');
56          require(reserveIn > 0 && reserveOut > 0, 'DAOLibrary: INSUFFICIENT_LIQUIDITY');
57          uint numerator = reserveIn.mul(amountOut).mul(10000);
58          uint denominator = reserveOut.sub(amountOut).mul(9970);
59          amountIn = (numerator / denominator).add(1);
60      }
```

Listing 3.1: DAOLibrary.sol

```
173     function swap(uint amount0Out, uint amount1Out, address to) external lock {
174         require(amount0Out > 0  amount1Out > 0, 'DAO: INSUFFICIENT_OUTPUT_AMOUNT');
175         (uint112 _reserve0, uint112 _reserve1,) = getReserves();
176         // gas savings
177         require(amount0Out < _reserve0 && amount1Out < _reserve1, 'DAO:
                INSUFFICIENT_LIQUIDITY');
178
179         uint balance0;
180         uint balance1;
181         {// scope for _token{0,1}, avoids stack too deep errors
182             address _token0 = token0;
183             address _token1 = token1;
184             require(to != _token0 && to != _token1, 'DAO: INVALID_TO');
185             if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out);
186             // optimistically transfer tokens
187             if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out);
188             // optimistically transfer tokens
189             // remove flash swap
190             // if (data.length > 0) IDAOCallee(to).daoCall(msg.sender, amount0Out,
                    amount1Out, data);
191             balance0 = IERC20(_token0).balanceOf(address(this));
192             balance1 = IERC20(_token1).balanceOf(address(this));
193         }
194         uint amount0In = balance0 > _reserve0 - amount0Out ? balance0 - (_reserve0 -
                amount0Out) : 0;
195         uint amount1In = balance1 > _reserve1 - amount1Out ? balance1 - (_reserve1 -
                amount1Out) : 0;
196         require(amount0In > 0  amount1In > 0, 'DAO: INSUFFICIENT_INPUT_AMOUNT');
197         {// scope for reserve{0,1}Adjusted, avoids stack too deep errors
198             uint balance0Adjusted = (balance0.mul(10000).sub(amount0In.mul(25)));
199             uint balance1Adjusted = (balance1.mul(10000).sub(amount1In.mul(25)));
200             require(balance0Adjusted.mul(balance1Adjusted) >= uint(_reserve0).mul(
                    _reserve1).mul(10000 ** 2), 'DAO: K');
201         }
202
203         _update(balance0, balance1, _reserve0, _reserve1);
204         emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to);
```

```
205        }
```

Listing 3.2:   DAOPair:swap()

Recommendation   Revisit the above mentioned routines to use the same swap fee rate in the whole protocol.

Status   The issue has been fixed in this commit: `457652a`.

3.2   Improved Sanity Checks For System Parameters

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium

- Target: `SwapToEarn`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The `SatoshiIncognito DaoSwap` protocol provides the swap transaction mining to the user and rewards airdrop to the referrer. All the token rewards can be added by the `owner` via the `SwapToEarn::addTokenRewards()` routine. While examining the airdrop rewards to the inviter, we notice the lack of proper sanity checks for the avaliable `DAO` token in the contract which may lead to the trading to be reverted.

To elaborate, we show below the code snippet of the `swapCall()` routine. As the name indicates, it is called from the swap router during a trading in the `DaoSwap`. Specifically, if the referrer (`inviter`) is provided during the `DAO` trade, the referrer will be rewarded with 1/1000 of the traded `DAO` (line 160-164). However, before transferring the `DAO` reward to the referrer, it does not validate if there is enough `DAO` avaliable in the contract. As a result, if the avaliable `DAO` is not enough, the transfer may fail (line 167), thus reverting the trade. With that, we suggest to validate the avaliable `DAO` balance. If the `DAO` reward amount is larger than the avaliable `DAO` balance, we can reset the `DAO` reward amount to the current avaliable `DAO` balance.

```
87      function swapCall(Params memory p) external {
88          address user = p.user;
89          address pair = p.pair;
90          address inviter = p.inviter;
91          address input = p.input;
92          address output = p.output;
93          uint256 amountIn = p.amountIn;
94          uint256 amountOut = p.amountOut;
95
96          require(msg.sender == swapRouter, "not swap router");
```

```
 97            if (!pairHasReward[pair]) {
 98                return;
 99            }
100
101            // input token has swap rewards
102            if (tokenRewards[input] > 0) {
103                uint256 d = uint256(IERC20Metadata(input).decimals());
104                uint256 staked = daoStakedInfo[user].staked / (10 ** d);
105                //efficiency based 10000
106                uint256 efficiency = 10;
107                if (staked < 200) {
108                    efficiency = 10;
109                } else if (staked >= 200 && staked < 500) {
110                    efficiency = 20;
111                } else if (staked >= 500 && staked < 1000) {
112                    efficiency = 30;
113                } else if (staked >= 1000 && staked < 5000) {
114                    efficiency = 40;
115                } else if (staked >= 5000) {
116                    efficiency = 60;
117                }
118                uint256 amount = amountIn.mul(efficiency).div(10000);
119                if (amount > tokenRewards[input]) {
120                    amount = tokenRewards[input];
121                }
122                tokenRewards[input] = tokenRewards[input].sub(amount);
123                swapRewards[user][input] = swapRewards[user][input].add(amount);
124                IERC20(input).safeTransfer(user, amount);
125            }
126
127            // output token has swap rewards
128            if (tokenRewards[output] > 0) {
129
130                uint256 d = uint256(IERC20Metadata(output).decimals());
131                uint256 staked = daoStakedInfo[user].staked / (10 ** d);
132                //efficiency based 10000
133                uint256 efficiency = 10;
134                if (staked < 200) {
135                    efficiency = 10;
136                } else if (staked >= 200 && staked < 500) {
137                    efficiency = 20;
138                } else if (staked >= 500 && staked < 1000) {
139                    efficiency = 30;
140                } else if (staked >= 1000 && staked < 5000) {
141                    efficiency = 40;
142                } else if (staked >= 5000) {
143                    efficiency = 60;
144                }
145
146                uint256 amount = amountOut.mul(efficiency).div(10000);
147                if (amount > tokenRewards[output]) {
148                    amount = tokenRewards[output];
```

```
149              }
150              tokenRewards[output] = tokenRewards[output].sub(amount);
151              swapRewards[user][output] = swapRewards[user][output].add(amount);
152              IERC20(output).safeTransfer(user, amount);
153          }
154
155          //inviter
156          if (inviter != address(0)) {
157              // based 10000
158              uint256 reward = 10;
159              uint256 amount = 0;
160              if (input == address(DAO)) {
161                  amount = amountIn.mul(reward).div(10000);
162              } else if (output == address(DAO)) {
163                  amount = amountOut.mul(reward).div(10000);
164              }
165              if (amount > 0) {
166                  totalInviteRewards = totalInviteRewards.add(amount);
167                  DAO.safeTransfer(inviter, amount);
168                  inviteRewards[inviter] = inviteRewards[inviter].add(amount);
169              }
170          }
171      }
```

<div align="center">Listing 3.3: <code>SwapToEarn::swapCall()</code></div>

Recommendation   Add proper sanity check in the above mentioned routine to reward the referrer with a reasonable and avaliable amount of `DAO`.

Status   The issue has been fixed in this commit: `457652a`.

3.3   Incompatibility with Deflationary Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `SwapToEarn`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As described in Section 3.1, the `SatoshiIncognito DaoSwap` protocol implements the swap transaction mining to the trader and rewards airdrop to the referrer. All the token rewards are added by the `owner` to this contract. Accordingly, the contract implements a number of low-level helper routines to transfer assets in or out of the contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual

token balances maintained in individual ERC20 token contract. In the following, we show the code snippet of the `addTokenRewards()` routine.

```
73    function addTokenRewards(address token, uint256 rewardsAmount) public onlyOwner {
74        require(rewardsAmount > 0, "amount: 0");
75        IERC20(token).safeTransferFrom(address(msg.sender), address(this), rewardsAmount
             );
76        tokenRewards[token] = tokenRewards[token].add(rewardsAmount);
77    }
```

Listing 3.4: `SwapToEarn::addTokenRewards()`

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these asset-transferring routines. In other words, the above operations, such as `addTokenRewards()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the contract before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into the `SwapToEarn`. In fact, the `SwapToEarn` is indeed in the position to effectively regulate the set of assets that can be listed. Meanwhile, there exist certain assets that may exhibit control switches that can be dynamically exercised to convert into deflationary.

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

Status   The issue has been fixed in this commit: `457652a`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: `Multiple contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `DaoSwap` protocol, there is a privileged account, i.e., `owner`, that plays a critical role in governing and regulating the system-wide operations (e.g., set the `DAO` address, set stake-lockup seconds). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `SwapToEarn` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

Specifically, the privileged functions in `SwapToEarn` allow for the `owner` to set the `DAO` address, set the `swapRouter`, set stake lock seconds, enable/disable the reward for a pair, and remove rewards from the contract, etc.

```
95      function init(address _dao, address _swapRouter) public onlyOwner {
96          swapRouter = _swapRouter;
97          DAO = IERC20(_dao);
98          unstakeWithdrawSeconds = 365 * 24 * 60 * 60;
99      }
100
101     function setUnstakeWithdrawSeconds(uint256 _seconds) public onlyOwner {
102         unstakeWithdrawSeconds = _seconds;
103     }
104
105     function setPairHasReward(address pair, bool hasReward) public onlyOwner {
106         require(pairHasReward[pair] != hasReward, "already set");
107         pairHasReward[pair] = hasReward;
108     }
109
110     function addTokenRewards(address token, uint256 rewardsAmount) public onlyOwner {
111         require(rewardsAmount > 0, "amount: 0");
112         IERC20(token).safeTransferFrom(address(msg.sender), address(this), rewardsAmount
                );
113         tokenRewards[token] = tokenRewards[token].add(rewardsAmount);
114     }
115
116     function removeTokenRewards(address token, uint256 rewardsAmount) public onlyOwner {
117         require(rewardsAmount > 0, "amount: 0");
118         require(rewardsAmount <= tokenRewards[token], "amount: not good");
119         tokenRewards[token] = tokenRewards[token].sub(rewardsAmount);
120         IERC20(token).safeTransfer(address(msg.sender), rewardsAmount);
```

```
121        }
```

Listing 3.5:   Example Privileged Operations in the `SwapToEarn` Contract

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the `owner` may also be a counter-party risk to the protocol users. It is worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation   Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status   This issue has been mitigated as the team confirms that the `DaoSwap` contracts will be handed over to the community for multi-signature co-management.

3.5   Potential Arbitrage Swaps To Drain All Rewards

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium

- Target: `SwapToEarn`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As described in Section 3.1, the `SwapToEarn` contract implements the swap transaction mining to the trader to reward the token trading in the pool. The rewards are carried out by staking `DAO` into this contract. And the user is rewarded in the input token or the output token if any of them has swap rewards added by the `owner`.

To elaborate, we show below the code snippet of the `swapCall()` routine. When the reward is enabled for the trade pair, and the input/output token has reward avaliable, the trader is rewarded in one of the five levels defined per the amount of `DAO` he/she stakes in the contract. The more `DAO` he/she stakes, the higher mining reward he/she can get. The five mining reward levels are defined as below (lines 106-117):

- Stake $0 - 199$ `DAO`, mining reward is 1/1000

- Stake $200 - 499$ `DAO`, mining reward 2/1000

- stake 500 − 999 DAO, mining reward 3/1000

- stake 1000 − 4999 DAO, mining reward 4/1000

- stake 5000 or more DAO, mining reward is 6/1000

The swap transaction mining could greatly incentivize users to trade in DaoSwap. However, it comes to our attention that, when the mining reward (efficiency) is bigger than the swap fee rate (25/10000), malicious trader may trigger repeated swaps to drain all the avaliable rewards once they are added by the owner. The swap transaction mining introduces big arbitrage chance, because the trader can get more rewards than he/she pays for the swap fee. As a result, other traders can never get reward from their normal trading in DaoSwap.

```
87     function swapCall(Params memory p) external {
88         address user = p.user;
89         address pair = p.pair;
90         address inviter = p.inviter;
91         address input = p.input;
92         address output = p.output;
93         uint256 amountIn = p.amountIn;
94         uint256 amountOut = p.amountOut;
95
96         require(msg.sender == swapRouter, "not swap router");
97         if (!pairHasReward[pair]) {
98             return;
99         }
100
101         // input token has swap rewards
102         if (tokenRewards[input] > 0) {
103             uint256 d = uint256(IERC20Metadata(input).decimals());
104             uint256 staked = daoStakedInfo[user].staked / (10 ** d);
105             //efficiency based 10000
106             uint256 efficiency = 10;
107             if (staked < 200) {
108                 efficiency = 10;
109             } else if (staked >= 200 && staked < 500) {
110                 efficiency = 20;
111             } else if (staked >= 500 && staked < 1000) {
112                 efficiency = 30;
113             } else if (staked >= 1000 && staked < 5000) {
114                 efficiency = 40;
115             } else if (staked >= 5000) {
116                 efficiency = 60;
117             }
118             uint256 amount = amountIn.mul(efficiency).div(10000);
119             if (amount > tokenRewards[input]) {
120                 amount = tokenRewards[input];
121             }
122             tokenRewards[input] = tokenRewards[input].sub(amount);
123             swapRewards[user][input] = swapRewards[user][input].add(amount);
```

```
124             IERC20(input).safeTransfer(user, amount);
125         }
126
127         // output token has swap rewards
128         if (tokenRewards[output] > 0) {
129
130             uint256 d = uint256(IERC20Metadata(output).decimals());
131             uint256 staked = daoStakedInfo[user].staked / (10 ** d);
132             //efficiency based 10000
133             uint256 efficiency = 10;
134             if (staked < 200) {
135                 efficiency = 10;
136             } else if (staked >= 200 && staked < 500) {
137                 efficiency = 20;
138             } else if (staked >= 500 && staked < 1000) {
139                 efficiency = 30;
140             } else if (staked >= 1000 && staked < 5000) {
141                 efficiency = 40;
142             } else if (staked >= 5000) {
143                 efficiency = 60;
144             }
145
146             uint256 amount = amountOut.mul(efficiency).div(10000);
147             if (amount > tokenRewards[output]) {
148                 amount = tokenRewards[output];
149             }
150             tokenRewards[output] = tokenRewards[output].sub(amount);
151             swapRewards[user][output] = swapRewards[user][output].add(amount);
152             IERC20(output).safeTransfer(user, amount);
153         }
154         ...
155     }
```

Listing 3.6: SwapToEarn:swapCall()

Recommendation   Revisit the swap transaction mining logic to prevent potential arbitrage swaps.

Status   The issue has been confirmed by the team that: For the time being, only DAO transaction mining is supported, and others are to be supported in future version. The DAO token transfer is set with a handling fee of 0.3%, and in DaoSwap there is also a swap fee of 0.3%, so at most the user can get ~0.1% profit which is consistent with the design to incentivise the using of DaoSwap.

3.6   Proper Management For The Invitation Rewards

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium

- Target: `SwapToEarn`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The `SatoshiIncognito DaoSwap` protocol provides an invitation reward in `DAO` to incentivize the inviter for the referral of `DaoSwap`. Per design, the rewards are supplied from the project team by directly transferring `DAO` to the contract. The inviter is rewarded during the `DAO` trading with 0.1% of the traded `DAO` which are directly transferred to the inviter from the contract (line 167). However, it comes to our attention that, besides the invitation rewards supplied by the project team, the `DAO` held in the contract may also come from users staking of `DAO`. Specially, if the `DAO` rewards supplied by the project team have been consumed, it may continue to consume users staking. As a result, user may fail to unstake the staking from the contract. Based on this, it is suggested to add an independent state variable to properly manage the invitation rewards to avoid the consuming of users staking.

```
87      function swapCall(Params memory p) external {
88          address user = p.user;
89          address pair = p.pair;
90          address inviter = p.inviter;
91          address input = p.input;
92          address output = p.output;
93          uint256 amountIn = p.amountIn;
94          uint256 amountOut = p.amountOut;
95
96          require(msg.sender == swapRouter, "not swap router");
97          if (!pairHasReward[pair]) {
98              return;
99          }
100
101         // input token has swap rewards
102         if (tokenRewards[input] > 0) {
103             uint256 d = uint256(IERC20Metadata(input).decimals());
104             uint256 staked = daoStakedInfo[user].staked / (10 ** d);
105             //efficiency based 10000
106             uint256 efficiency = 10;
107             if (staked < 200) {
108                 efficiency = 10;
109             } else if (staked >= 200 && staked < 500) {
110                 efficiency = 20;
111             } else if (staked >= 500 && staked < 1000) {
112                 efficiency = 30;
```

```
113              } else if (staked >= 1000 && staked < 5000) {
114                  efficiency = 40;
115              } else if (staked >= 5000) {
116                  efficiency = 60;
117              }
118              uint256 amount = amountIn.mul(efficiency).div(10000);
119              if (amount > tokenRewards[input]) {
120                  amount = tokenRewards[input];
121              }
122              tokenRewards[input] = tokenRewards[input].sub(amount);
123              swapRewards[user][input] = swapRewards[user][input].add(amount);
124              IERC20(input).safeTransfer(user, amount);
125          }
126
127          // output token has swap rewards
128          if (tokenRewards[output] > 0) {
129
130              uint256 d = uint256(IERC20Metadata(output).decimals());
131              uint256 staked = daoStakedInfo[user].staked / (10 ** d);
132              //efficiency based 10000
133              uint256 efficiency = 10;
134              if (staked < 200) {
135                  efficiency = 10;
136              } else if (staked >= 200 && staked < 500) {
137                  efficiency = 20;
138              } else if (staked >= 500 && staked < 1000) {
139                  efficiency = 30;
140              } else if (staked >= 1000 && staked < 5000) {
141                  efficiency = 40;
142              } else if (staked >= 5000) {
143                  efficiency = 60;
144              }
145
146              uint256 amount = amountOut.mul(efficiency).div(10000);
147              if (amount > tokenRewards[output]) {
148                  amount = tokenRewards[output];
149              }
150              tokenRewards[output] = tokenRewards[output].sub(amount);
151              swapRewards[user][output] = swapRewards[user][output].add(amount);
152              IERC20(output).safeTransfer(user, amount);
153          }
154
155          //inviter
156          if (inviter != address(0)) {
157              // based 10000
158              uint256 reward = 10;
159              uint256 amount = 0;
160              if (input == address(DAO)) {
161                  amount = amountIn.mul(reward).div(10000);
162              } else if (output == address(DAO)) {
163                  amount = amountOut.mul(reward).div(10000);
164              }
```

```
165              if (amount > 0) {
166                  totalInviteRewards = totalInviteRewards.add(amount);
167                  DAO.safeTransfer(inviter, amount);
168                  inviteRewards[inviter] = inviteRewards[inviter].add(amount);
169              }
170          }
171      }
```

Listing 3.7: `SwapToEarn::swapCall()`

Recommendation Add an independent state variable to properly manage the invitation rewards to avoid the consuming of users staking.

Status The issue has been mitigated by the project team who will closely monitor the consumed invitation rewards (`totalInviteRewards`) and supply `DAO`s accordingly to keep users staking safe.

4 | Conclusion

In this audit, we have analyzed the `DaoSwap` design and implementation. `DaoSwap` is a professional trading platform on `BSC`, which is initiated and managed by the `DaoSwap Foundation`. The protocol supports users to freely create trading pairs and earn transaction fees to achieve decentralized token exchange and automatic market making. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

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

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

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

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

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

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