SMART CONTRACT AUDIT REPORT

for

Poop

Prepared By: Xiaomi Huang

PeckShield

June 3, 2023

Document Properties

Client	Poop
Title	Smart Contract Audit Report
Target	Poop
Version	1.0
Author	Elizabeth Wu
Auditors	Elizabeth Wu, Xuxian Jiang
Reviewed by	Elizabeth Wu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 3, 2023	Elizabeth Wu	Final Release
1.0-rc	May 30, 2023	Elizabeth Wu	Release Candidate

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 `Poop` protocol design document and related smart contract source code, 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 branch of `Poop` protocol 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 Poop Staking

`Poop Finance` is a `DeFi` protocol that recycles the shitcoins into `POOP`. The protocol extends the template of `ERC20` token from `Openzeppelin` by adding two functions, which are `buy()` and `sell()` . These functions will serve as interface to buy `POOP` with native token and sell `POOP` to native token. Both of the functions charge fees when buying and selling through the functions. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Poop

Item	Description
Name	Poop
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 3, 2023

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

- https://github.com/poop-finance/poop-contract/tree/security (c41289b)

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

- https://github.com/poop-finance/poop-contract/tree/security (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

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

Likelihood

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 contract is considered safe regarding the check item. For any discovered issue, we might further

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

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) [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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Poop` 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

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

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 1 critical-severity vulnerability, 1 medium-severity vulnerability, and 1 low-severity vulnerability.

Table 2.1: Key Poop Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Critical	Arbitrary Setting of User Referral	Business Logic	Fixed
PVE-002	Low	Safe-Version Replacement With safeApprove()	Coding Practices	Fixed
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Arbitrary Setting of User Referral

- ID: PVE-001
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `Poop`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned before, the `POOP` token forks from `Openzeppelin`'s `ERC20` template and adds two interfaces, e.g., `buy()` and `sell()`. At the same time, these functions also serve as the interface to set `referral` address. While review the `referral` address configuration logic, we notice a critical bug which will allow anyone to set the referral address of any user arbitrarily. To elaborate, we show below the `buy()` routine.

```
    function buy(address receiver, address referral) external payable nonReentrant {
        require(START, "not started");
        require(PUBLIC_BUY  whitelist[msg.sender], "illegal caller");
        require(msg.value >= MIN_BUY_AMOUNT && msg.value <= MAX_BUY_AMOUNT, "must trade
            over min and below max");

        address currentReferral = referrals[receiver];
        if (currentReferral == address(0) && referral != address(0)) {
            referrals[receiver] = referral;
            currentReferral = referral;
            emit ReferralRelation(receiver, referral, block.timestamp);
        }
        if (currentReferral == address(0)) {
            currentReferral = INCENTIVE_VAULT;
        }

        // Mint Poop to sender
        uint256 poop = ETHtoPOOP(msg.value);
        _mint(receiver, (poop * BUY_AFTER_FEE) / FEE_BASE);
```

```
        // Reserve fee
        uint value = msg.value;
        if (RESERVE_FEE_ADDRESS != address(0)) {
            sendEth(RESERVE_FEE_ADDRESS, value / RESERVE_FEES);
        }
        // Referral Fee
        if (currentReferral != address(0)) {
            sendEth(currentReferral, value / REFERRAL_FEE);
            emit ReferralReward(receiver, currentReferral, value / REFERRAL_FEE, block.
                timestamp);
        }

        emit Price(block.timestamp, poop, msg.value);
    }
```

Listing 3.1: `Poop::buy()`

It comes to our attention that this routine does not properly handle the validation of `msg.sender` and `receiver`, which will allow any `msg.sender` to set any `referrals[receiver]`. Also, the address can not be changed once configured. As a result, a malicious actor could front run every `buy()` transaction to set the referral address which could make a profit in the `buy()` transaction.

Recommendation Revise the `buy()` logic to properly validate `receiver == msg.sender`.

Status This issue has been fixed in this commit: `1148f70`.

3.2 Safe-Version Replacement With safeApprove()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PoopRouter`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

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. The `approve()` function does not have a return value. However, the `IERC20` interface has defined the following `approve()` interface with a `bool` return value: `function approve(address spender, uint256 amount)external returns (bool)`. As a result, the call to `approve()` may expect a return value. With the lack of return value of `USDT`'s `approve()`, the call will be unfortunately reverted.

```
    /**
    * @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 3.2: USDT Token **Contract**

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. To use this library you can add a using SafeERC20 for IERC20.

While reviewing the current `PoopRouter` contract, it comes to our attention that while `safeTransfer()/safeTransferFrom()` has been used, the `approve()` is still being used instead of the safe version `safeApprove()`

In the following, we show the `_swapExactTokens()` routine in the `PoopRouter` contract. If the `USDT` token is supported as `token`, the unsafe version of `token.approve(address(router), balance)` (line 88 and 95) may revert as there is no return value in the `USDT` token contract's `approve()` implementation (but the `IERC20` interface expects a return value)!

```
77     function _swapExactTokens(IUniswapV2Router02 router, address[] calldata path, uint amount, uint minAmount, address referral) internal nonReentrant {
78         IUniswapV2Factory factory = IUniswapV2Factory(router.factory());
79         require(address(factory) != address(0), "router not support");

81         require(path.length >= 2 && path[path.length - 1] == address(WETH), "illegal path");
82         require(minAmount >= MIN_AMOUNT, "illegal amount");

84         IERC20 token = IERC20(path[0]);
85         token.safeTransferFrom(msg.sender, address(this), amount);

87         uint balance = token.balanceOf(address(this));
88         token.approve(address(router), balance);
89         try router.swapExactTokensForETHSupportingFeeOnTransferTokens(balance, minAmount, path, address(this), block.timestamp) {
```

```
            //we defined minAmount for swap, so there would be values;
            WETH.withdraw(WETH.balanceOf(address(this)));
        } catch {
            //do nothing
        }
        token.approve(address(router), 0);

        balance = token.balanceOf(address(this));
        if (balance > 0) {
            token.safeTransfer(VAULT, balance);
        }

        require(address(this).balance >= MIN_BUY_AMOUNT, "illegal buy amount");
        POOP.buy{value: address(this).balance}(msg.sender, referral);
    }
```

Listing 3.3: `PoopRouter::_swapExactTokens()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`.

Status This issue has been fixed in this commit: `918a871`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Staking`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Poop` token contract and related protocols, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the protocol-wide operations (e.g., configure protocol parameters). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged `owner` account and its related privileged accesses in current contract.

To elaborate, we show below the related function. The `setVault()` routine supports the configuration of `VAULT` value, which takes `ETH/BNB` and buys/sells tokens.

```
    function setVault(address vault) external onlyOwner {
        require(vault != address(0), "illegal vault");
        VAULT = vault;
```

```
    }
```

Listing 3.4: `PoopRouter::setVault()`

We understand the need of the privileged functions for contract maintenance, but 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 The issue has been confirmed by the team.

4 Conclusion

In this audit, we have analyzed the `Poop` protocol design and implementation. `Poop Finance` is a `DeFi` protocol that recycles the shitcoins into `POOP`. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

[1] MITRE. CWE-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.