SMART CONTRACT AUDIT REPORT

for

PinkSale SubscriptionPool

Prepared By: Xiaomi Huang

PeckShield

October 8, 2022

Document Properties

Client	PinkSale
Title	Smart Contract Audit Report
Target	PinkSale SubscriptionPool
Version	1.0
Author	Stephen Bie
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 8, 2022	Stephen Bie	Final Release
1.0-rc	September 21, 2022	Shulin Bie	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 design document and related smart contract source code of PinkSale SubscriptionPool, 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 PinkSale SubscriptionPool

PinkSale is a decentralized launchpad that allows users to launch their own token and create their own initial token sale. In order to provide a safe environment for all of the investors that use the PinkSale ecosystem, PinkSale implements strict KYC procedures to deter dangerous and deceptive behavior. The audited SubscriptionPool is one of the supported presale formats. In the subscription format, users can commit an amount of assets (e.g., BNB) towards a token sale, where their final allocation of the new token is determined by the ratio of their committed assets against the total committed assets by all participating users.

Table 1.1: Basic Information of PinkSale SubscriptionPool

Item	Description
Target	PinkSale SubscriptionPool
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	October 8, 2022

In the following, we show the MD5 hash value of the file used in this audit:

- MD5 (SubscriptionPool-09072022.sol) = c0412b28893e668f6f318da170b55514

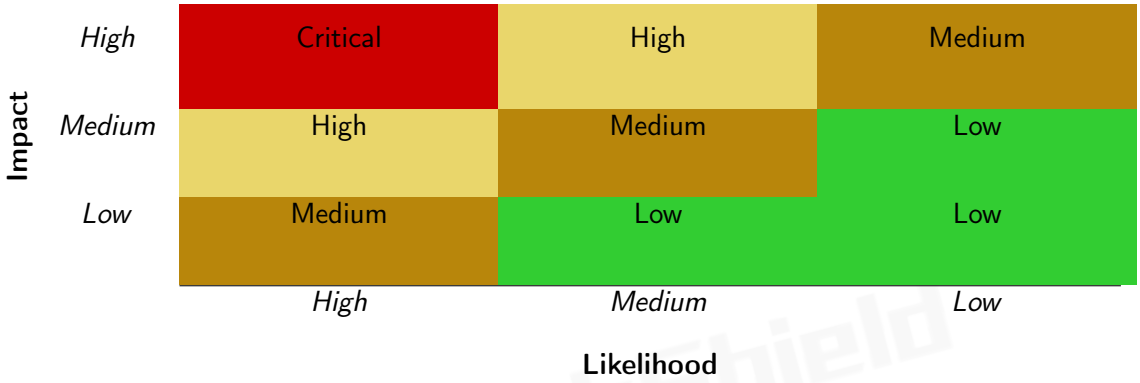
And here is the final MD5 hash value of the compressed file after all fixes for the issues found in the audit have been checked in:

- MD5 (SubscriptionPool-09292022.sol) = 799c117f7ad1a77be97441522aeb6e03

1.2 About PeckShield

PeckShield Inc. [10] 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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

- 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.

Table 1.3: The Full List of Check Items

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

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

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

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [8], 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 `PinkSale SubscriptionPool` 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	0	
High	0	
Medium	1	■
Low	2	■■
Informational	1	■
Total	4	

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 medium-severity vulnerability, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key PinkSale SubscriptionPool Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incompatibility with Deflationary/Rebasing Tokens	Business Logic	Confirmed
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-003	Informational	Suggested Event Generation for Key Operations	Coding Practices	Confirmed
PVE-004	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 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SubscriptionPool`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the `PinkSale SubscriptionPool` implementation, the `SubscriptionPool` contract is the main entry for interaction with users. In particular, one entry routine, i.e., `contributeCustomCurrency()`, is designed to commit an amount of assets (e.g., `USDC`) towards a token presale. Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the `SubscriptionPool` 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 contracts.

```
3200    function contributeCustomCurrency(uint256 amount) external {
3201        require(poolSettings.currency != address(0), "Invalid currency");
3202        IERC20(poolSettings.currency).safeTransferFrom(
3203            msg.sender,
3204            address(this),
3205            amount
3206        );
3207        _contribute(amount);
3208    }
```

Listing 3.1: `SubscriptionPool::contributeCustomCurrency()`

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 certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not

meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `contributeCustomCurrency()`, 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 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 contract 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 `PinkSale SubscriptionPool`. In the `PinkSale SubscriptionPool` implementation, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., `USDT`) that may have control switches that can be dynamically exercised to suddenly become one.

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

Status The issue has been confirmed by the team. The team decides to leave it as is considering there is no need to support deflationary/rebasing token.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SubscriptionPool/LaunchPadLibrary`
- Category: Coding Practices [6]
- 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 analyze 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.

```
194     /**
195     * @dev Approve the passed address to spend the specified amount of tokens on behalf
            of msg.sender.
196     * @param _spender The address which will spend the funds.
197     * @param _value The amount of tokens to be spent.
198     */
199     function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

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

207         allowed[msg.sender][_spender] = _value;
208         Approval(msg.sender, _spender, _value);
209     }
```

Listing 3.2: USDT Token **Contract**

It is important to note that 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.

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.

In the following, we show the `LaunchPadLibrary::addLiquidity()` routine. If the `USDT` token is supported as `token`, the unsafe version of `IERC20(token).approve(address(router), liquidityToken)` (line 2693) may revert as there is no return value in the `USDT` token contract's `approve()` implementation (but the `IERC20` interface expects a return value)!

```
2684    function addLiquidity(
2685        IUniswapV2Router02 router,
2686        address currency,
2687        address token,
2688        uint256 liquidityCurrency,
2689        uint256 liquidityToken,
2690        uint256 rate,
2691        address pool
2692    ) public returns (uint256 liquidity) {
2693        IERC20(token).approve(address(router), liquidityToken);
2694        IUniswapV2Pair pair = getPair(router, currency, token);
2695        uint256 minCurrency = liquidityCurrency;
2696        uint256 minToken = liquidityToken;
2697        ...
```

```
2698
2699        if (currency == address(0)) {
2700            (, , liquidity) = router.addLiquidityETH{value: liquidityCurrency}(
2701                token,
2702                liquidityToken,
2703                minToken,
2704                minCurrency,
2705                pool,
2706                block.timestamp
2707            );
2708        } else {
2709            IERC20(currency).approve(address(router), type(uint256).max);
2710            (, , liquidity) = router.addLiquidity(
2711                token,
2712                currency,
2713                liquidityToken,
2714                liquidityCurrency,
2715                minToken,
2716                minCurrency,
2717                pool,
2718                block.timestamp
2719            );
2720        }
2721    }
```

Listing 3.3: `LaunchPadLibrary::addLiquidity()`

Note another routine, i.e., `SubscriptionPool::_lockLiquidity()`, shares the same issue.

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

Status The issue has been addressed by the team in the latest version: 799c117f7ad1a77be97441522aeb6e03.

3.3 Suggested Event Generation for Key Operations

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `SubscriptionPool`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in

transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

While examining the events that reflect the protocol dynamics, we notice there are several key operations that lack meaningful events to reflect their changes. In the following, we show several representative routines.

```
3552     function updateCalculatedData(
3553         uint256 totalVolumePurchased,
3554         address[] memory users,
3555         uint256[] memory amounts
3556     ) external onlyGovernance {
3557         require(users.length == amounts.length, "Invalid length");
3558         poolStates.totalVolumePurchased = totalVolumePurchased;
3559         for (uint256 i = 0; i < users.length; i++) {
3560             purchasedOf[users[i]] = amounts[i];
3561         }
3562     }
3563
3564     function setCalculationStage(Stage stage) external onlyGovernance {
3565         calculationStage.stage = stage;
3566     }
3567
3568     function setCanFinalize() external onlyGovernance {
3569         calculationStage.stage = Stage.CALCULATED;
3570     }
```

Listing 3.4: SubscriptionPool

With that, we suggest to emit meaningful events for these key operations. Also, the key event information is better indexed. Note each emitted event is represented as a topic that usually consists of the signature (from a keccak256 hash) of the event name and the types (uint256, string, etc.) of its parameters. Each indexed type will be treated like an additional topic. If an argument is not indexed, it will be attached as data (instead of a separate topic). Considering that the key information is typically queried, it is better treated as a topic, hence the need of being indexed.

Recommendation Properly emit the above-mentioned events with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status The issue has been confirmed by the team.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: SubscriptionPool
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the PinkSale SubscriptionPool implementation, there is a privileged governance account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). In the following, we show the representative functions potentially affected by the privilege of the governance account.

```
3552    function updateCalculatedData(
3553        uint256 totalVolumePurchased,
3554        address[] memory users,
3555        uint256[] memory amounts
3556    ) external onlyGovernance {
3557        require(users.length == amounts.length, "Invalid length");
3558        poolStates.totalVolumePurchased = totalVolumePurchased;
3559        for (uint256 i = 0; i < users.length; i++) {
3560            purchasedOf[users[i]] = amounts[i];
3561        }
3562    }
3563    function setCalculationStage(Stage stage) external onlyGovernance {
3564        calculationStage.stage = stage;
3565    }
3566    function setCanFinalize() external onlyGovernance {
3567        calculationStage.stage = Stage.CALCULATED;
3568    }
```

Listing 3.5: SubscriptionPool

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a DAO-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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 PinkSale SubscriptionPool design and implementation. PinkSale is a decentralized launchpad that allows users to launch their own token and create their own initial token sale. The audited SubscriptionPool is one of the supported presale formats. In the subscription format, users can commit an amount of assets (e.g., BNB) towards a token sale, where their final allocation of the new token is determined by the ratio of their committed assets against the total committed assets by all participating users. 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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

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

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

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

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

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