SECURITY AUDIT REPORT

for

PinkLock Solana

Prepared By: Xiaomi Huang

PeckShield

April 1, 2024

Document Properties

Client	PinkSale
Title	Security Audit Report
Target	PinkLock Solana
Version	1.0
Author	Daisy Cao
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 1, 2024	Daisy Cao	Final Release
1.0-rc	March 28, 2024	Daisy Cao	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 PinkLock Solana contract, 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 PinkLock Solana

PinkSale is a decentralized launchpad, securing its position as the leading platform for users to initiate their own tokens and orchestrate personalized initial token sales, all without the requirement of coding expertise. PinkLock Solana allows users to edit tokenomic chart on their presale. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of PinkLock Solana

Item	Description
Name	PinkSale
Website	https://pinksale.finance
Type	Solana
Language	Rust
Audit Method	Whitebox
Latest Audit Report	April 1, 2024

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

- https://github.com/pinkmoonfinance/pink-lock-rust (8545c59a.)

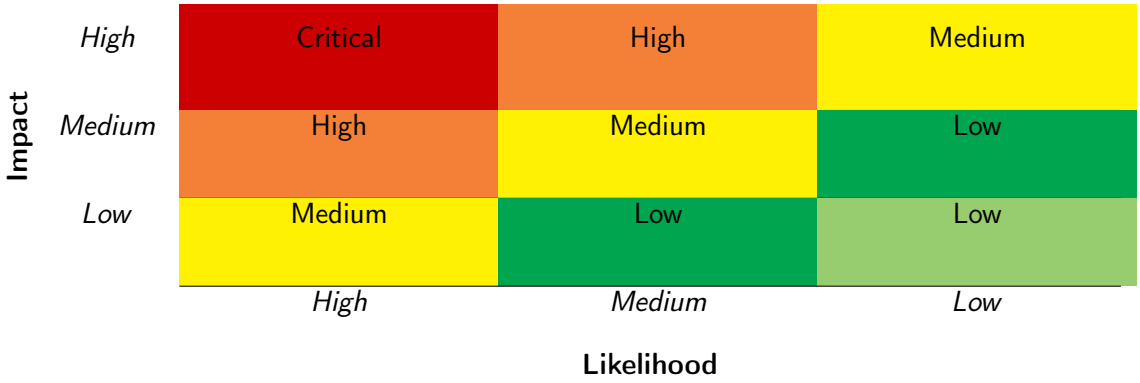
And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/pinkmoonfinance/pink-lock-rust (c17f4264.)

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



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 accordingly classified into four categories, 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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 `PinkLock Solana` implementations. 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	2	■■
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 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 2 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Missing Lock Amount Validation in initialize_locker	Business Logic	Confirmed
PVE-002	Medium	Lack of Account Check in EditLocker	Business Logic	Fixed
PVE-003	Low	Redundant State/Code Removal	Business Logic	Fixed
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 Missing Lock Amount Validation in initialize_locker

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `initialize_locker.rs`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

`PinkLock Solana` allows users to create a locker via the `initialize_locker()` function. The `SPL` token amount to be locked should be transfered from the user's account. Therefore, the quantity in the user's account should be greater than the intended lock quantity.

In the following, we show the related code snippet of the `initialize_locker()` routine. Our analysis shows that the intended validation is missing.

```
1   pub fn initialize_locker(
2       ctx: Context<InitializeLocker>,
3       _nonce_timestamp: u64,
4       owner: Pubkey,
5       tge_or_unlock_date: u64,
6       cycle: u32,
7       tge_bps: u16,
8       cycle_bps: u16,
9       amount: u64,
10      title: String,
11  ) -> Result<()> {
12      require!(amount > 0, ErrorCode::InvalidLockAmount);
13      let lock_date = Clock::get()?.unix_timestamp as u64;
14      require!(tge_or_unlock_date > lock_date, ErrorCode::InvalidUnlockDate);
15
16      if tge_bps > 0 || cycle_bps > 0 || cycle > 0 {
17          require!(
18              (tge_bps > 0 && tge_bps < BPS_DENOMINATOR)
19                  && (cycle_bps > 0 && cycle_bps < BPS_DENOMINATOR)
```

```
20                      && (tge_bps + cycle_bps <= BPS_DENOMINATOR),
21                  ErrorCode::InvalidBps
22              );
23              require!(cycle > 0, ErrorCode::InvalidCycle);
24          }
25
26          require!(title.len() < TITLE_LENGTH.into(), ErrorCode::InvalidTitle);
27
28          if ctx.accounts.locker_manager.lock_fee != 0 {
29              transfer_sol(
30                  &ctx.accounts.signer,
31                  &ctx.accounts.fee_receiver,
32                  &ctx.accounts.system_program,
33                  ctx.accounts.locker_manager.lock_fee,
34              )?;
35          }
36      ...
37  }
```

Listing 3.1: `initialize_locker::initialize_locker()`

Recommendation Validate the input amount in the above `initialize_locker()` routine.

Status This issue has been confirmed.

3.2 Lack of Account Check in EditLocker

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `edit_locker.rs`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

By design, `PinkLock Solana` allows the `locker` owner to modify the associated locker information, including the amount. When the locked amount is increased, there is a need to transfer additional funds from the user's account to the vault. While examining the topup logic, we notice the associated implementation can be improved.

In the following, we show the code snippet of the related `EditLocker` data structure. The `user_token_account` is the account passed in by the user, and its mint amount must match the mint of the `SPL` token that will be locked, i.e., `constraint = user_token_account.mint == mint.key()`. Apparently, this constraint is currently missing.

```
100     pub struct EditLocker<'info> {
```

```
101     #[account(mut)]
102     pub locker: Box<Account<'info, Locker>>,
103
104     #[account(
105         constraint = mint.key() == locker.mint,
106         token::token_program = token_program,
107     )]
108     pub mint: Box<InterfaceAccount<'info, Mint>>,
109
110     #[account(mut, constraint = locker_vault.key() == locker.vault)]
111     pub locker_vault: Box<InterfaceAccount<'info, TokenAccount>>,
112
113     #[account(
114         mut,
115         constraint = locker_vault.key() == locker.vault && locker_vault.mint == locker.
                mint,
116         token::token_program = token_program,
117     )]
118     pub user_token_account: Box<InterfaceAccount<'info, TokenAccount>>,
119     ...
120 }
```

Listing 3.2: The `edit_locker::EditLocker` Structure

Recommendation Add the `mint` validation in the above `EditLocker` data structure.

Status The issue has been fixed by this commit: c17f426.

3.3 Redundant State/Code Removal

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `sanity.rs`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

As mentioned in Section 3.2, the `edit_locker()` function is used to modify the locker information. The locker is initialized by saving the corresponding vault, so it is necessary to ensure that the passed-in vault matches the one saved in the locker. While examining current implementation, we observe the inclusion of certain unused code or the presence of unnecessary redundancy that can be removed.

To elaborate, we show below the definition of the related `EditLocker` data structure. The vault validation imposes the same constraint `locker_vault.key()== locker.vault)` (lines 210 and 215) in

both accounts: `user_token_account` and `locker_vault`. Apparently, the same constraint has been enforced twice and we only need to enforce once.

```
200     pub struct EditLocker<'info> {
201     #[account(mut)]
202     pub locker: Box<Account<'info, Locker>>,
203
204     #[account(
205         constraint = mint.key() == locker.mint,
206         token::token_program = token_program,
207     )]
208     pub mint: Box<InterfaceAccount<'info, Mint>>,
209
210     #[account(mut, constraint = locker_vault.key() == locker.vault)]
211     pub locker_vault: Box<InterfaceAccount<'info, TokenAccount>>,
212
213     #[account(
214         mut,
215         constraint = locker_vault.key() == locker.vault && locker_vault.mint == locker.
                mint,
216         token::token_program = token_program,
217     )]
218     pub user_token_account: Box<InterfaceAccount<'info, TokenAccount>>,
219     ...
220     }
```

Listing 3.3: The `edit_locker::EditLocker` Structure

Recommendation Consider the removal of the redundant code with a simplified, consistent implementation.

Status The issue has been fixed by this commit: c17f426.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In `PinkLock Solana`, there is a privileged account, i.e., `locker_manager`. This account plays a critical role in governing and regulating the system-wide operations (e.g., set lock fee, whitelist_programs etc.). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `update_locker_manager` contract as an example and show the representative functions potentially affected by the privileges of the `admin` account.

```
300     pub fn update_locker_manager(
301         ctx: Context<UpdateLockerManager>,
302         lock_fee: u64,
303         fee_receiver: Pubkey,
304         whitelist_programs: Vec<Pubkey>,
305     ) -> Result<()> {
306         let locker_manager = &mut ctx.accounts.locker_manager;
307         locker_manager.update(lock_fee, fee_receiver, whitelist_programs)
308     }
```

Listing 3.4: `update_locker_manager::update_locker_manager()`

We understand the need of the privileged functions for proper `PinkLock Solana` operations, but at the same time the extra power to the `locker_manager` may also be a counter-party risk to the `PinkLock Solana` contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the list of extra privileges granted to `PinkLock Solana` explicit to `PinkLock Solana` contract users.

Status This issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the `PinkLock Solana` contract design and implementation. 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

[3] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.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.