SMART CONTRACT AUDIT REPORT

for

AVA Daily Earn Lockup

Prepared By: Xiaomi Huang

PeckShield

November 27, 2025

Document Properties

Client	AVA
Title	Smart Contract Audit Report
Target	AVA Lockup
Version	1.0.1
Author	Xuxian Jiang
Auditors	Matthew Jiang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0.1	November 27, 2025	Xuxian Jiang	Post-Final Release #1
1.0	November 9, 2025	Xuxian Jiang	Final Release
1.0-rc1	November 8, 2025	Xuxian Jiang	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 Daily Earn Lockup contract in AVA, 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 AVA Daily Earn Lockup

The Daily Earn Lockup support in AVA manages a token lockup and reward system with multiple membership tiers. Users may lock ERC20 tokens for fixed periods to earn rewards, with higher membership tiers offering better terms and possible NFT-based bonus rates. It supports delayed and immediate withdrawals (the latter incurring a small fee), as well as administrative controls for pausing, user activation/deactivation, and updating membership parameters. The basic information of AVA Lockup is as follows:

Table 1.1: Basic Information of AVA Lockup

Item	Description
Issuer	AVA
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 27, 2025

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

- https://github.com/AVA-Foundation/ava-lockup-contracts.git (8f00bdd)

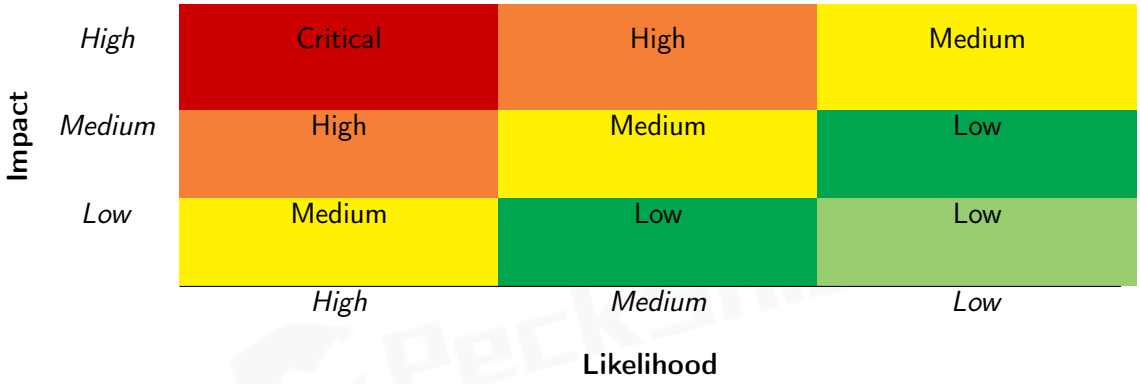
And here is the commit ID after all fixes for the issues found in the audit have been checked in. The audited `Daily Earn Lockup` contract has an `Ethereum` deployment at `0x5CC235Eca665ED2A752802ed784EAe373e6B0Beb`, which has its owner and community fee wallet configured to `0x58653987Ff3837ADBE6383F670f6935fcDE521b0` and `0xE234857A497deCf6239911C8190c195a0eaBB638`, respectively.

- https://github.com/AVA-Foundation/ava-lockup-contracts.git (d807a5f)

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 the 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.

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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

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 checklist of 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) [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

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 Logic	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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the Daily Earn Lockup contract in AVA. 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	2	■■
Low	2	■■
Informational	0	
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 AVA Lockup Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Possibly Missing Last Lockup Timestamp Update	Business Logic	Resolved
PVE-002	Low	Possibly Double Rewards From Multiple User Deactivation	Business Logic	Resolved
PVE-003	Low	Timely Reward Claims Upon Possible APR Change	Coding Practices	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Possibly Missing Last Lockup Timestamp Update

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

Description

For each user, the audited token lockup contract maintains the last lockup timestamp to accurately calculate the accrued rewards. While examining the logic to update the last lockup timestamp, we notice an issue that may miss the timely update when a new lockup is added or a previous one is canceled.

To elaborate, we show below the implementation of the related `lockup()` routine. This routine is invoked when a new lockup is added. In particular, if the user has a previous lockup, the earned rewards may be claimed. However, our analysis shows that if the earned rewards amount is `0`, the last timestamp is not updated (line 599). As a result, the new lockup is effectively recorded with the previous timestamp as the latest timestamp, which has undesirable implications in calculating the earned rewards for the new lockup. To fix, there is a need to always update the new lockup timestamp with `block.timestamp` when earned rewards are claimed.

```
268     function lockup(uint256 lockedAmount) external whenNotPaused nonReentrant {
269         _validateLockupConditions(lockedAmount, msg.sender);
271         // Transfer lockup tokens from the walletAddress to the contract.
272         lockupToken.safeTransferFrom(msg.sender, address(this), lockedAmount);
274         string memory userId = _getUserId(msg.sender);
276         // If the walletAddress already has a lockup, send him his earned tokens
277         // before increasing the lockedAmount.
278         if (userIdToLockupStats[userId].lockedAmount > 0) {
```

```
279             _claimReward(userId, msg.sender);
280         } else {
281             // For a newbie, initialize lastTimeStamp to current time.
282             userIdToLockupStats[userId].lastTimeStamp = block.timestamp;
283         }

285         // Increase the lockedAmount.
286         userIdToLockupStats[userId].lockedAmount += lockedAmount;

288         // Update the total locked amount.
289         totalLockedAmount += lockedAmount;

291         emit EvtLockup(msg.sender, lockedAmount);
292     }
```

Listing 3.1: `DailyEarnLockUp::lockup()`

```
575     function _claimReward(
576         string memory userId,
577         address walletAddress
578     ) private returns (uint256) {
579         // De-activated user cannot get any earn
580         if (userIdToLockupStats[userId].deactivated) {
581             return 0;
582         }

584         (
585             uint256 earn,
586             uint256 timeElapsedSinceLastClaim
587         ) = getEarnAmountFromLockupStats(walletAddress);
588         if (earn > 0) {
589             // Update lastTimeStamp to the timestamp rounded up to the last full day.
590             LockupStats storage stats = userIdToLockupStats[userId];
591             stats.lastTimeStamp += timeElapsedSinceLastClaim;
592             stats.earnedAmount += earn;

594             lockupToken.safeTransfer(walletAddress, earn);
595             emit EvtClaim(walletAddress, earn);
596             return earn;
597         } else {
598             // If no earn, return 0.
599             return 0;
600         }
601     }
```

Listing 3.2: `DailyEarnLockUp::_claimReward()`

Furthermore, when a previous withdrawal request is canceled, we shall need to update the lockup timestamp with `block.timestamp` as well.

Recommendation Revise the above-mentioned routines to properly update the user's lockup timestamp when the earned rewards are timely calculated and claimed.

Status This issue has been fixed in the following commit: `feaf135`.

3.2 Possibly Double Rewards From Multiple User Deactivation

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `DailyEarnLockUp`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

To manage the eligibility of participating users, the lockup contract allows the authorized managers to activate or deactivate users on a need basis. When a user is deactivated, the rewards are timely accumulated up to the deactivation timestamp. Our analysis shows the rewards from withdrawal requests may be repeatedly calculated for each deactivation.

To elaborate, we show below the implementation of the related `deactivateUser()` routine. It has a rather straightforward logic in claiming accrued rewards and marking the deactivation state. Note the rewards-claiming helper routine `_claimReward()` (line 617) does timely update the last lockup timestamp so that the rewards will be properly calculated. But another helper routine `_claimRewardFromWithdrawRequest()` (line 619) does not, which may lead to double rewards if a user is deactivated once, next activated, and then deactivated again.

```
608     function deactivateUser(string memory userId) external isAuthorized {
609         require(bytes(userId).length > 0, "Invalid userId");
610         require(
611             userIdToLockupStats[userId].deactivated == false,
612             "userId already deactivated"
613         );
615         // Auto send any accrued rewards before deactivation.
616         address walletAddress = userIdToWalletAddress[userId];
617         _claimReward(userId, walletAddress);
619         _claimRewardFromWithdrawRequest(walletAddress);
621         userIdToLockupStats[userId].deactivated = true;
623         emit EvtDeactivateUser(userId);
624     }
```

Listing 3.3: `DailyEarnLockUp::deactivateUser()`

Recommendation Revise the above `deactivateUser()` routine to ensure rewards are always correctly accumulated when a user is deactivated.

Status This issue has been fixed in the following commit: a11a999.

3.3 Timely Reward Claims Upon Possible APR Change

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: DailyEarnLockUp
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

As mentioned earlier, the audited contract manages the token lockup and reward with multiple membership tiers and each membership tier may be configured with various reward rates. When a user's membership tier or related parameters are adjusted, there is a need to timely accumulate user rewards before applying the new settings. While reviewing current setter routines, we notice two specific ones need to be improved.

To elaborate, we show below the implementation of one setter routine, i.e., updateUserIdToAmountNFTs(). As the name indicates, this routine is used to updated the number of NFTs for the given user id. When a user's NFT amount is updated, the rewards may be affected. With that, we need to accumulate the rewards before updating the user's NFT amount. Similarly, when the user's membership tier or type is updated (via updateUserIdToMembershipType()), we also need to accumulate rewards for both locked amount and withdraw-requested amount.

```
701     function updateUserIdToAmountNFTs(
702         string calldata userId,
703         uint256 amountNFTs
704     ) external isAuthorized {
705         require(bytes(userId).length > 0, "Invalid userId");
706         require(
707             userIdToMembershipType[userId] == MembershipType.SmartDiamond,
708             "Not SmartDiamond membership type"
709         );
711         userIdToAmountNFTs[userId] = amountNFTs;
713         emit EvtUpdateUserIdToAmountNFTs(userId, amountNFTs);
714     }
```

Listing 3.4: DailyEarnLockUp::updateUserIdToAmountNFTs()

Recommendation Revise the above-mentioned setter routines to properly accrue rewards from both locked amount and requested amount for withdrawal.

Status This issue has been fixed in the following commit: `d807a5f`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `DailyEarnLockUp`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the audited `DailyEarnLockUp` contract, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the lockup-wide operations (e.g., configure withdrawal fee, manage authorizers, and withdraw contract funds). It also has the privilege to control or govern the flow of assets within the protocol contracts (e.g., perform the emergency withdrawal). In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
755     function withdrawByAdmin(
756         address recipient,
757         uint256 amount
758     ) external isOwner {...}
759
760     function takeImmediateWithdrawalFeeCollected(
761         address recipient,
762         uint256 amount
763     ) external isOwner {...}
764
765     /**
766      * @notice Set the membership type for a specific wallet address.
767      */
768     function setMembershipTypeToCondition(
769         MembershipType membershipType,
770         LockupCondition memory lockupCondition
771     ) external isOwner {...}
772
773     /**
774      * @notice Set the immediate withdrawal fee
775      */
776     function setImmediateWithdrawalFee(
777         uint256 _immediateWithdrawalFee
778     ) external isOwner {...}
779
780     /**
781      * @notice Set the community wallet address
782      */
```

```
783     function setCommunityWallet(address _communityWallet) external isOwner {...}

785     /**
786      * @notice Set the maximum total locked amount.
787      */
788     function setMaxTotalLockedAmount(
789         uint256 _maxTotalLockedAmount
790     ) external isOwner {...}

792     function updateCommonMinLockupAmount(
793         uint256 newMinLockupAmount
794     ) external isOwner {...}

796     function updateCommonWithdrawPeriod(
797         uint256 newWithdrawPeriodInSeconds
798     ) external isOwner {...}
```

Listing 3.5: Example Privileged Operations in DailyEarnLockUp

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the 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 Promptly transfer the privileged account to the intended DAO-like governance contract. All changes 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

4 Conclusion

In this audit, we have analyzed the design and implementation of the Daily Earn Lockup contract in AVA. It is a token lockup and reward system with multiple membership tiers. Users may lock ERC20 tokens for fixed periods to earn rewards, with higher membership tiers offering better terms and possible NFT-based bonus rates. It supports delayed and immediate withdrawals (the latter incurring a small fee), as well as administrative controls for pausing, user activation/deactivation, and updating membership parameters. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.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.