



SMART CONTRACT AUDIT REPORT

for

AVA Daily Earn Lockup



Prepared By: Xiaomi Huang

PeckShield
September 30, 2026

Document Properties

Client	AVA
Title	Smart Contract Audit Report
Target	AVA Lockup
Version	1.0
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	September 30, 2026	Xuxian Jiang	Final Release
1.0-rc1	September 28, 2026	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About AVA Daily Earn Lockup	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Revisited Reward Calculation Upon Associated NFT Amount Update	11
3.2	Trust Issue of Admin Keys	12
4	Conclusion	15
	References	16

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	September 30, 2026

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> (b5604e3)

And here is the live address deployed on `Ethereum` as well as the corresponding commit hash for the deployment.

- <https://etherscan.io/address/0x5CC235Eca665ED2A752802ed784EAe373e6B0Beb>
- <https://github.com/AVA-Foundation/ava-lockup-contracts.git> (b5604e3)

1.2 About PeckShield

PeckShield Inc. [7] 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	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [6]:

- 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 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

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) [5], 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 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.

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	1	
Low	1	
Informational	0	
Total	2	

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

Table 2.1: Key AVA Lockup Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Revisited Reward Calculation Upon Associated NFT Amount Update	Business Logic	Acknowledged
PVE-002	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 Revisited Reward Calculation Upon Associated NFT Amount Update

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: DailyEarnLockUp
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

A user may request to retrieve the previously locked tokens with delayed or immediate withdrawals. The request may be cancelled and the cancellation still refunds the user the entitled rewards. While reviewing the reward calculation logic, we notice an issue that may be resulted from two (privileged) functions that allows to change the associated NFT number for a given user.

To elaborate, we use one privileged function, i.e., `updateUserIdToAmountNFTs()`, as an example and show below its implementation. This function has a rather straightforward logic in updating a given user's associated NFT number (`amountNFTs`). Before updating the given user with a new NFT number, this function invokes `_claimReward()` first to claim any accrued rewards under the current NFT number. However, if the given user has a pending withdrawal request and the new NFT number is larger than current one, the user may choose to cancel the withdraw request so that she may claim more rewards with the use of the larger new NFT number! Note this issue also affects another privileged function – `updateUserIdToMembershipType()`.

```
777     function updateUserIdToAmountNFTs(  
778         string calldata userId,  
779         uint256 amountNFTs  
780     ) external isAuthorized {  
781         require(bytes(userId).length > 0, "Invalid userId");  
782         require(  
783             userIdToMembershipType[userId] == MembershipType.SmartDiamond,
```

```

784         "Not SmartDiamond membership type"
785     );
786     require(
787         userIdToAmountNFTs[userId] != amountNFTs,
788         "UserId already has this amountNFTs"
789     );

791     // In case the userId is updated with a new amountNFTs, claimReward first with
792     // the current amountNFTs.
793     // This ensures that any rewards accrued under the current amountNFTs are
794     // claimed before changing it.
795     address walletAddress = userIdToWalletAddress[userId];
796     _claimReward(userId, walletAddress);

797     userIdToAmountNFTs[userId] = amountNFTs;

798     emit EvtUpdateUserIdToAmountNFTs(userId, amountNFTs);
799 }

```

Listing 3.1: DailyEarnLockUp::updateUserIdToAmountNFTs()

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 revisit the reward calculation logic so that the user only claims the entitled rewards.

Status This issue has been confirmed. While the contract itself is non-upgradeable, the impact from this issue is strictly bounded: at most +1.5% APR, a 200k token cap per user, over the withdrawal wait period. Moreover, it may only be triggered with the authorized accounts that are well-managed and are by design trustworthy.

3.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: DailyEarnLockUp
- Category: Security Features [3]
- 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.

```

844     function withdrawByAdmin(
845         address recipient,
846         uint256 amount
847     ) external isOwner {...}

849     function takeImmediateWithdrawalFeeCollected(
850         address recipient,
851         uint256 amount
852     ) external isOwner {...}

854     /**
855      * @notice Set the membership type for a specific wallet address.
856      */
857     function setMembershipTypeToCondition(
858         MembershipType membershipType,
859         LockupCondition memory lockupCondition
860     ) external isOwner {...}

862     /**
863      * @notice Set the immediate withdrawal fee
864      */
865     function setImmediateWithdrawalFee(
866         uint256 _immediateWithdrawalFee
867     ) external isOwner {...}

869     /**
870      * @notice Set the community wallet address
871      */
872     function setCommunityWallet(address _communityWallet) external isOwner {...}

874     /**
875      * @notice Set the maximum total locked amount.
876      */
877     function setMaxTotalLockedAmount(
878         uint256 _maxTotalLockedAmount
879     ) external isOwner {...}

881     function updateCommonMinLockupAmount(
882         uint256 newMinLockupAmount
883     ) external isOwner {...}

885     function updateCommonWithdrawPeriod(
886         uint256 newWithdrawPeriodInSeconds
887     ) external isOwner {...}

```

Listing 3.2: 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 The issue has been mitigated as the team makes use of a multi-sig account as the owner. In particular, on Ethereum, the owner account is 3-of-4 Safe, deployed at <https://etherscan.io/address/0x58653987Ff3837ADBE6383F670f6935fcDE521b0>.

Further, the contract supports the so-called authorized role that can update a user's registered wallet address, which directly affects who can withdraw that user's funds. The team clarifies that the authorized key is held in AWS KMS, and ownership sits with the 3-of-4 Safe, which can revoke it and pause the contract.



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 addressed.

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



References

- [1] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [2] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [3] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [4] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [5] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [6] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [7] PeckShield. PeckShield Inc. <https://www.peckshield.com>.