SMART CONTRACT AUDIT REPORT

for

Instadapp Avocado

Prepared By: Xiaomi Huang

PeckShield

January 14, 2023

Document Properties

Client	Instadapp
Title	Smart Contract Audit Report
Target	Avocado
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 14, 2023	Luck Hu	Final Release
1.0-rc	January 4, 2023	Luck Hu	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 the Avocado protocol, 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 Avocado

Instadapp is a DeFi portal that aggregates a variety of major protocols using a smart wallet layer, making it easy for users to make the best decisions about their assets and execute previously complex transactions seamlessly. The audited Avocado protocol is an important component of the Instadapp ecosystem. Avocado enables a fluid and seamless way to execute web3 interactions by enabling multi-network gas and account abstraction. The basic information of Avocado is as follows:

Table 1.1: Basic Information of Avocado

Item	Description
Target	Avocado
Website	https://instadapp.io/
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	January 14, 2023

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

- https://github.com/Instadapp/avocado-contracts.git (8388e007)

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

- https://github.com/Instadapp/avocado-contracts.git (a7102d92)

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

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on 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.

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

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

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.

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 `Avocado` 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	1	🟦
Informational	1	🟦
Total	3	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Avocado Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Suggested Immutable Usage for Gas Efficiency	Coding Practices	Fixed
PVE-002	Informational	Redundant State/Code Removal	Coding Practices	Fixed
PVE-003	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 Suggested Immutable Usage for Gas Efficiency

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `AvoFactory`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1099 [1]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

In the following, we show the key state variable `_avoSafeBytecode` in the `AvoFactory` contract, which can only be assigned with `keccak256(abi.encodePacked(type(AvoSafe).creationCode))` (line 121). If there is no need to dynamically update this key variable, it can be declared as either constant or `immutable` for gas efficiency. In particular, the above state variable can be defined as `immutable` and initialized in the `constructor()`, as the `type(AvoSafe).creationCode` will not change after the `constructor()` .

```
120   function updateAvoSafeBytecode() public returns (bytes32 newAvoSafeBytecode_) {
121     newAvoSafeBytecode_ = keccak256(abi.encodePacked(type(AvoSafe).creationCode));
122     _avoSafeBytecode = newAvoSafeBytecode_;
```

```
    return newAvoSafeBytecode_;
}
```

Listing 3.1: `AvoFactory::updateAvoSafeBytecode()`

Recommendation Revisit the state variable definition and make extensive use of `immutable` states for gas efficiency.

Status This issue has been fixed in the following commit: `a7102d92`.

3.2 Redundant State/Code Removal

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `AvoVersionsRegistry`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1099 [1]

Description

By design, the `AvoVersionsRegistry` contract maintains a list of valid versions of the various `Avo`-related contracts. While examining its logic, we observe the inclusion of certain unused code or the presence of unnecessary redundancy that can be improved or simplified.

To elaborate, we show below the related code snippets of the contract. Firstly, it defines a list of errors at the beginning of the contract. However, we notice the error `AvoVersionsRegistry__Unauthorized()` (line 14) is not used anywhere in the contract. Hence it could be removed safely.

```
13    contract AvoVersionsRegistry is IAvoVersionsRegistry, Initializable,
          OwnableUpgradeable {
14        error AvoVersionsRegistry__Unauthorized();
15        error AvoVersionsRegistry__InvalidParams();
16        error AvoVersionsRegistry__InvalidVersion();
17        ...
18    }
```

Listing 3.2: `AvoVersionsRegistry.sol`

Moreover, in the `initialize()` routine, there is a validation against the input `owner_` to ensure `owner_ != address(0)` (line 75). We notice the validation could be simplified by directly using the `validAddress(owner_)` modifier. What's more, it initializes the contract owner by invoking the `__Ownable_init()` routine first (line 79), which sets the owner to the `msg.sender`. Then it invokes the `transferOwnership(owner_)` to transfer the ownership to the input `owner_` (line 80). However, the ownership initialization could be simplified by calling the `_transferOwnership(owner_)` directly to transfer the ownership to the input `owner_`.

```
    function initialize(address owner_) public initializer {
        if (owner_ == address(0)) {
            revert AvoVersionsRegistry__InvalidParams();
        }

        __Ownable_init();
        transferOwnership(owner_);
    }

    modifier validAddress(address _address) {
        if (_address == address(0)) {
            revert AvoVersionsRegistry__InvalidParams();
        }
        _;
    }
```

Listing 3.3: `AvoVersionsRegistry::initialize()`

Recommendation Consider the removal of unused code and simplify the redundant code.

Status The issue has been fixed in this commit: `ff94689b`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `AvoVersionsRegistry`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [2]

Description

In the `Avocado` protocol, there is a privileged account, i.e., `owner`, that plays a critical role in governing and regulating the protocol-wide operations (e.g., set the valid versions for the avoWallet). In the following, we show the representative functions potentially affected by the privileges of the `owner` account.

Specifically, the privileged functions in the `AvoVersionsRegistry` contract allow for the `owner` to set the `avoFactory` to create new `AvoSafe` instances and set the valid versions for the `AvoWallet/avoForwarder` which could be used to upgrade the implementations for the `AvoWallet/avoForwarder`.

```
    /// @notice              sets the current AvoFactory address
    /// @param avoFactory_   address of the new avoFactory
    function setAvoFactory(address avoFactory_) external onlyOwner validAddress(
        avoFactory_) {
        avoFactory = IAvoFactory(avoFactory_);
    }

    /// @notice                 sets the status for a certain address as valid AvoWallet
         version
    /// @param avoWallet_       the address of the contract to treat as AvoWallet
        version
    /// @param allowed_         flag to set this address as valid version (true) or not
        (false)
    /// @param setDefault_      flag to indicate whether this version should
        automatically be set as new
    ///                         default version for new deployments at the linked
        AvoFactory
    function setAvoWalletVersion(
        address avoWallet_,
        bool allowed_,
        bool setDefault_
    ) external onlyOwner validAddress(avoWallet_) {
        if (!allowed_ && setDefault_) {
            // can't be not allowed but supposed to be set as default
            revert AvoVersionsRegistry__InvalidParams();
        }

        avoWalletVersions[avoWallet_] = allowed_;

        if (setDefault_) {
            // register the new version as default version at the linked AvoFactory
            avoFactory.setAvoWalletImpl(avoWallet_);
        }

        emit SetAvoWalletVersion(avoWallet_, allowed_, setDefault_);
    }

    /// @notice                 sets the status for a certain address as valid
        AvoForwarder (proxy) version
    /// @param avoForwarder_    the address of the contract to treat as AvoForwarder
        version
    /// @param allowed_         flag to set this address as valid version (true) or not
        (false)
    function setAvoForwarderVersion(address avoForwarder_, bool allowed_)
        external
        onlyOwner
        validAddress(avoForwarder_)
    {
        avoForwarderVersions[avoForwarder_] = allowed_;

        emit SetAvoForwarderVersion(avoForwarder_, allowed_);
    }
```

Listing 3.4: `AvoVersionsRegistry.sol`

We emphasize that the privilege assignment is indeed necessary and consistent with the protocol design. However, it is worrisome if the privileged account is a plain EOA account. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Note that a compromised privileged 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 This issue has been mitigated as the team confirmed that they will use multi-sig to manage the owner.

4 Conclusion

In this audit, we have analyzed the `Avocado` design and implementation. `Instadapp` is a `DeFi` portal that aggregates a variety of major protocols using a smart wallet layer, making it easy for users to make the best decisions about their assets and execute previously complex transactions seamlessly. The audited `Avocado` protocol is an important component of the `Instadapp` ecosystem. `Avocado` enables a fluid and seamless way to execute web3 interactions by enabling multi-network gas and account abstraction. 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-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

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

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

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