SMART CONTRACT AUDIT REPORT

for

Instadapp Avocado (V3)

Prepared By: Xiaomi Huang

PeckShield

June 12, 2023

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	June 12, 2023	Xuxian Jiang	Final Release
1.0-rc	June 11, 2023	Jing Wang	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 (v3)` 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

`Avocado` is a powerful next-generation super wallet for web3. It greatly simplifies the web3 experience through advancements in account abstraction. `Avocado` abstracts gas, aggregates various `EVM` networks, and provides a smart wallet solution for interacting with current and future blockchains. The basic information of `Avocado` is as follows:

Table 1.1: Basic Information of Avocado

Item	Description
Target	Avocado
Website	https://avocado.instadapp.io/
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	June 12, 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/tree/feature/3.0.0-refactorings (95feb57)

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 (TBD)

1.2 About PeckShield

PeckShield Inc. [8] 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 \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [7]:

- 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 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) [6], 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 (v3)` 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	0	
Low	2	■ ■
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 2 low-severity vulnerabilities and 1 informational recommendation.

Table 2.1: Key Avocado Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Implicit Assumption of addSigners_-in Ascending Order	Business Logic	Confirmed
PVE-002	Low	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-003	Informational	Improved Sanity Checks of System/-Function Parameters	Coding Practices	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 Implicit Assumption of addSigners_ in Ascending Order

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: AvoSignersList
- Category: Coding Practices [5]
- CWE subcategory: CWE-1099 [1]

Description

The Avocado (v3) protocol has a built-in AvoSignersList contract, which is designed to keep track of allowed signers for AvoMultiSafes. It keeps a list of all signers linked to an AvoMultiSafe or all AvoMultiSafes for a certain signer address. While examining the current syncing logic, we notice the current implementation has an implicit assumption.

To elaborate, we show below the related syncAddAvoSignerMappings() function. As the name indicates, this function is designed to sync the added signers to AvoSignersList. The added signers are specified in the given addSigners_. While this function properly handles the signers-adding request, it has an implicit assumption that addSigners_ will be given in ascending order. As a result, if the new signers are given in non-ascending order, the addition will not be successful.

```
    function syncAddAvoSignerMappings(address avoMultiSafe_, address[] calldata addSigners_) external {
        // make sure avoMultiSafe_ is an actual AvoMultiSafe
        if (avoFactory.isAvoSafe(avoMultiSafe_) == false) {
            revert AvoSignersList__InvalidParams();
        }

        uint256 addSignersLength_ = addSigners_.length;
        if (addSignersLength_ == 1) {
            ...
        } else {
            // get actual signers present at AvoMultisig to make sure data here will be correct
```

```
            address[] memory allowedSigners_ = IAvoMultisigV3(avoMultiSafe_).signers();
            uint256 allowedSignersLength_ = allowedSigners_.length;
            // track last allowed signer index for loop performance improvements
            uint256 lastAllowedSignerIndex_;

            bool isAllowedSigner_; // keeping this variable outside the loop so it is
                not re-initialized in each loop -> cheaper
            for (uint256 i; i < addSignersLength_; ) {
                // because allowedSigners_ and addSigners_ must be ordered ascending the
                     for loop can be optimized each
                // new cycle to start from the position where the last signer has been
                    found
                for (uint256 j = lastAllowedSignerIndex_; j < allowedSignersLength_; ) {
                    if (allowedSigners_[j] == addSigners_[i]) {
                        isAllowedSigner_ = true;
                        lastAllowedSignerIndex_ = j + 1; // set to j+1 so that next
                            cycle starts at next array position
                        break;
                    }

                    // could be optimized by checking if allowedSigners_[j] >
                        recoveredSigners_[i] immediately skipping with a break;
                    // because that implies that the recoveredSigners_[i] can not be
                        present in allowedSigners_ due to sort.
                    // but that would optimize the failing invalid case and in turn
                        increase cost for the default case where
                    // the input data is valid -> skip.

                    unchecked {
                        ++j;
                    }
                }

                // validate signer trying to add mapping for is really allowed at
                    AvoMultisig
                if (!isAllowedSigner_) {
                    revert AvoSignersList__InvalidParams();
                }

                // reset isAllowedSigner_ for next loop
                isAllowedSigner_ = false;

                if (trackInStorage) {
                    // add method also checks if signer is already mapped to avocado
                        Multisig, returns false in that case
                    if (_safesPerSigner[addSigners_[i]].add(avoMultiSafe_) == true) {
                        emit SignerMappingAdded(addSigners_[i], avoMultiSafe_);
                    }
                    // else ignore silently if mapping is already present
                } else {
                    emit SignerMappingAdded(addSigners_[i], avoMultiSafe_);
                }
```

```
                unchecked {
                    ++i;
                }
            }
        }
    }
```

Listing 3.1: `AvoDepositManager::processWithdraw()`

Recommendation Revisit the above logic to handle the `addSigners_` in non-ascending order explicitly.

Status The issue has been confirmed. The team clarifies the method will fail with the error `AvoSignersList__InvalidParams()`. And it is acceptable to share the same error message when trying to add an invalid signer which is not allowed at `AvoMultisig`.

3.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [3]

Description

In the `Avocado (v3)` protocol, there is a privileged account, i.e., `owner`, that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure system parameters). In the following, we show the representative functions potentially affected by the privileges of the `owner` account.

Specifically, the privileged functions in the `AvoDepositManagerOwnerActions` contract allow for the `owner` to set the `withdrawAddress_` to receive all the `depositToken_`.

```
    function setWithdrawAddress(address withdrawAddress_) external onlyOwner
        validAddress(withdrawAddress_) {
        withdrawAddress = withdrawAddress_;
        emit SetWithdrawAddress(withdrawAddress_);
    }

    function withdraw() external {
        IERC20 depositToken_ = depositToken;
        uint256 withdrawLimit_ = withdrawLimit;

        uint256 balance_ = depositToken_.balanceOf(address(this));
```

```
        if (balance_ > withdrawLimit_) {
            uint256 withdrawAmount_;
            unchecked {
                // can not underflow because of if statement just above
                withdrawAmount_ = balance_ - withdrawLimit_;
            }

            depositToken_.safeTransfer(withdrawAddress, withdrawAmount_);
        }
    }
```

Listing 3.2: Example Privileged Functions in AvoVersionsRegistry

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.

3.3 Improved Sanity Checks of System/Function Parameters

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: AvoCoreConstantsOverride
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [2]

Description

In the Avocado (v3) protocol, the AvoCoreConstantsOverride contract is designed to define the immutables. While reviewing the implementation of the AvoCoreConstantsOverride contract, we notice that the constructor() function can benefit from additional sanity checks.

To elaborate, we show below the related code snippet of the AvoCoreConstantsOverride contract. Specifically, the current implementation fails to check the given argument authorizedMinFee_ is smaller than authorizedMaxFee_.

```
    constructor(
        string memory domainSeparatorName_,
        string memory domainSeparatorVersion_,
        uint256 castAuthorizedReserveGas_,
        uint256 castEventsReserveGas_,
        uint256 authorizedMinFee_,
        uint256 authorizedMaxFee_,
        address authorizedFeeCollector_,
        bool isMultisig
    ) {
        DOMAIN_SEPARATOR_NAME_HASHED = keccak256(bytes(domainSeparatorName_));
        DOMAIN_SEPARATOR_VERSION_HASHED = keccak256(bytes(domainSeparatorVersion_));

        CAST_AUTHORIZED_RESERVE_GAS = castAuthorizedReserveGas_;
        CAST_EVENTS_RESERVE_GAS = castEventsReserveGas_;

        // min & max fee settings, fee collector address are required
        if (authorizedMinFee_ == 0  authorizedMaxFee_ == 0  authorizedFeeCollector_ ==
            address(0)) {
            revert AvoCore__InvalidParams();
        }

        AUTHORIZED_MIN_FEE = authorizedMinFee_;
        AUTHORIZED_MAX_FEE = authorizedMaxFee_;
        AUTHORIZED_FEE_COLLECTOR = payable(authorizedFeeCollector_);

        IS_MULTISIG = isMultisig;
    }
```

Listing 3.3: `AvoCoreConstantsOverride::constructor()`

Recommendation Validate the input arguments by ensuring `authorizedMinFee_ < authorizedMaxFee_` in the above `constructor()` function.

Status The issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the `Avocado (v3)` design and implementation. `Avocado` is a powerful next-generation super wallet for web3. The audited `Avocado (v3)` protocol is an important component of the `Instadapp` ecosystem and is designed to enable 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.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 VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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