SMART CONTRACT AUDIT REPORT

for

AVault

Prepared By: Patrick Lou

PeckShield

April 5, 2022

Document Properties

Client	AVault Finance
Title	Smart Contract Audit Report
Target	AVault
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Shulin Bie, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 5, 2022	Xuxian Jiang	Final Release
1.0-rc1	April 2, 2022	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Patrick Lou
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 AVault 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 AVault

The Astar Vault (Avault) is designed to allow Astar users to deposit their assets safely and earn a high return. In essence, Avault is a yield aggregator platform that provides aLP/aToken to DeFi users with automated compounding yields at empirically optimal intervals while pooling gas fees through smart contracts and best yield optimization strategies. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The AVault Protocol

Item	Description
Issuer	AVault Finance
Website	https://avault.network
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 5, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this audit only covers the AVaultBase.sol contract.

- https://github.com/AVaultFinance/avault-contracts.git (692d276)

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

- https://github.com/AVaultFinance/avault-contracts.git (29417c9)

1.2 About PeckShield

PeckShield Inc. [11] 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

Impact

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- 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) [9], 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 design and implementation of the `AVault` protocol. 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	3	■■■
Informational	0	
Total	5	

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 medium-severity vulnerabilities and 3 low-severity vulnerabilities.

Table 2.1: Key AVault Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Proper Paths Update in setPaths()	Business Logic	Fixed
PVE-002	Low	Possible Sandwich/MEV For Reduced Swap Amount	Time And State	Mitigated
PVE-003	Low	Possible Costly LPs From Improper Vault Initialization	Time And State	Fixed
PVE-004	Low	Improper withdrawal Logic with Right _unfarm() Amount	Business Logic	Fixed
PVE-005	Medium	Trust on Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Proper Paths Update in setPaths()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `AVaultBase`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `AVaultBase` protocol is no exception. Specifically, if we examine the `AVaultBase` contract, it has defined a number of protocol-wide risk parameters, such as `buyBackRate` and `withdrawFeeFactor`. In the following, we show the corresponding routines that allow for their changes.

```
    function setPaths(
        address[] memory _earnedToWethPath,
        address[] memory _wethToAVAPath,
        address[] memory _earnedToToken0Path,
        address[] memory _earnedToToken1Path,
        address[] memory _token0ToEarnedPath,
        address[] memory _token1ToEarnedPath
    ) external virtual onlyOwner{
        require(earnedToWethPath[0] == _earnedToWethPath[0] && earnedToWethPath[
            earnedToWethPath.length - 1] == _earnedToWethPath[_earnedToWethPath.length -
             1], "earnedToWethPath");
        require(wethToAVAPath[0] == _wethToAVAPath[0] && wethToAVAPath[wethToAVAPath.
            length - 1] == _wethToAVAPath[_wethToAVAPath.length - 1], "wethToAVAPath");
        require(earnedToToken0Path[0] == _earnedToToken0Path[0] && earnedToToken0Path[
            earnedToToken0Path.length - 1] == _earnedToToken0Path[_earnedToToken0Path.
            length - 1], "earnedToToken0Path");
        require(earnedToToken1Path[0] == _earnedToToken1Path[0] && earnedToToken1Path[
            earnedToToken1Path.length - 1] == _earnedToToken1Path[_earnedToToken1Path.
            length - 1], "earnedToToken1Path");
```

```
        require(token0ToEarnedPath[0] == _token0ToEarnedPath[0] && token0ToEarnedPath[
            token0ToEarnedPath.length - 1] == _token0ToEarnedPath[_token0ToEarnedPath.
            length - 1], "token0ToEarnedPath");
        require(token1ToEarnedPath[0] == _token1ToEarnedPath[0] && token1ToEarnedPath[
            token1ToEarnedPath.length - 1] == _token1ToEarnedPath[_token1ToEarnedPath.
            length - 1], "token1ToEarnedPath");
        emit PathsUpdated();
    }
```

Listing 3.1: `AVaultBase::setPaths()`

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain setter functions do not properly update these parameters.

To elaborate, we show above the `setPaths()` routine, which is designed to configure various swap paths for token conversion. However, it comes to our attention the current setter only performs the necessary validation on the given parameters and does not properly save these configurations!

Recommendation Validate any changes regarding these system-wide parameters and properly save them in the storage. If necessary, also consider emitting relevant events for their changes.

Status The issue has been fixed by this commit: `29417c9`.

3.2 Possible Sandwich/MEV For Reduced Swap Amount

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `AVaultBase`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

As a yield aggregator protocol, the `Avault` protocol has the constant need of swapping one token to another. For example, the computed staking rewards may need to convert into a target token as the final payout. Our analysis shows this mechanism can be improved to avoid unnecessary reward loss.

To elaborate, we show below the related `_safeSwap()` routine. As the name indicates, the function performs a swap operation with the intended slippage control. Note that the token conversion essentially performs the swap according to the given swap path (line 528).

```
512     function _safeSwap(
513         address _uniRouterAddress,
514         uint256 _amountIn,
```

```
        uint256 _slippageFactor,
        address[] memory _path,
        address _to,
        uint256 _deadline
    ) internal virtual {
        uint256[] memory amounts =
            IPancakeRouter02(_uniRouterAddress).getAmountsOut(_amountIn, _path);
        uint256 amountOut = amounts[amounts.length.sub(1)];

        IPancakeRouter02(_uniRouterAddress)
            .swapExactTokensForTokensSupportingFeeOnTransferTokens(
            _amountIn,
            amountOut.mul(_slippageFactor).div(1000),
            _path,
            _to,
            _deadline
        );
    }
```

Listing 3.2: `AVaultBase::_safeSwap()`

We notice the conversion is routed to the external `_uniRouterAddress` without the ineffective slippage control. With that, it is possible for a malicious actor to launch a flashloan-assisted attack to claim the majority of swaps, resulting in a significantly less amount after the swap. This is possible if the `_safeSwap()` function suffers from a sandwich attack.

Recommendation Develop an effective mitigation to the above sandwich attack to better protect the interests of trading users.

Status The issue has been mitigated as the use of a withdraw fee limits the potential gain. Moreover, the team plans to trigger the reinvestment on a daily basis, which also limits the trade size.

3.3 Possible Costly LPs From Improper Vault Initialization

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `AVaultBase`
- Category: Time and State [6]
- CWE subcategory: CWE-362 [2]

Description

The `AVault` protocol allows users to deposit supported assets and get in return the share to represent the pool ownership. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the pool share extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the `deposit()` routine, which is used for participating users to deposit the supported assets and get respective pool shares in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
80     function deposit(address _userAddress, uint256 _wantAmt)
81         external
82         virtual
83         nonReentrant
84         whenNotPaused
85     {
86         IERC20().safeTransferFrom(
87             address(msg.sender),
88             address(this),
89             _wantAmt
90         );

92         uint256 sharesAdded = _wantAmt;
93         if (wantLockedTotal > 0 && totalSupply() > 0) {
94             sharesAdded = _wantAmt
95                 .mul(totalSupply())
96                 .div(wantLockedTotal);
97         }
98         _mint(_userAddress, sharesAdded);

100        if(isEarnable && _dice()){
101            _earn();
102        }
103        _farm();

105        updateWantLockedTotal();
106    }
```

Listing 3.3: `AVaultBase::deposit()`

Specifically, when the pool is being initialized (line 91), the share value directly takes the value of `sharesAdded` (line 92), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `sharesAdded = _wantAmt = 1 WEI`. With that, the actor can further deposit a huge amount of the underlying assets with the goal of making the pool share extremely expensive.

An extremely expensive pool share can be very inconvenient to use as a small number of 1 `Wei` may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current deposit logic to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure a guarded launch process that safeguards the first deposit to avoid being manipulated.

Status The issue has been resolved as the team plans to follow a guarded launch so that a trusted user will be the first to deposit.

3.4 Improper withdrawal Logic with Right _unfarm() Amount

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `AVaultBase`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned earlier, `Avault` is a yield aggregator platform that provides automated compounding yields to its users. While reviewing the related withdrawal logic, we notice the current implementation needs to be improved.

To elaborate, we show below the related `withdraw()` function. This function implements a rather straightforward logic in computing the amount for withdrawal based on the given share amount. However, since certain assets are invested in external strategies, it is likely that the holding amount may not be sufficient to meet the withdraw request. For that, there is a need to withdraw the missing amount from the strategies. Our analysis on the calculation of the missing amount shows the current implementation simply calls `_unfarm(_wantAmt)` to withdraw the full request amount, not the missing amount `_wantAmt - wantAmt`.

```
    function withdraw(address _userAddress, uint256 _shareAmount)
        external
        virtual
        nonReentrant
    {
        uint _wantAmt = wantLockedTotal * _shareAmount / totalSupply();
        if (withdrawFeeFactor < withdrawFeeFactorMax) {
            _wantAmt = _wantAmt.mul(withdrawFeeFactor).div(
                withdrawFeeFactorMax
            );
        }
```

```
        require(_wantAmt > 0, "_wantAmt == 0");
        burn(_shareAmount);

        if(isEarnable && _dice()){
            _earn();
        }

        uint256 wantAmt = IERC20(wantAddress).balanceOf(address(this));
        if(wantAmt < _wantAmt){
            _unfarm(_wantAmt);
            wantAmt = IERC20(wantAddress).balanceOf(address(this));
        }

        if (_wantAmt > wantAmt) {
            _wantAmt = wantAmt;
        }

        IERC20(wantAddress).safeTransfer(_userAddress, _wantAmt);

        _farm();
        updateWantLockedTotal();
    }
```

Listing 3.4: `AVaultBase::withdraw()`

Recommendation Revise the above `withdraw()` function with the proper amount to withdraw from the strategies.

Status The issue has been fixed by this commit: `29417c9`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `AVaultBase`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

In the `AVault` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the system-wide operations (e.g., configure various settings). 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.

```
    function pause() public virtual onlyOwner {
        _pause();
    }

    function unpause() public virtual onlyOwner {
        _unpause();
    }

    function setSettings(
        uint256 _withdrawFeeFactor,
        uint256 _buyBackRate,
        uint256 _slippageFactor
    ) public virtual onlyOwner {
        require(
            _withdrawFeeFactor >= withdrawFeeFactorLL,
            "_withdrawFeeFactor too low"
        );
        require(
            _withdrawFeeFactor <= withdrawFeeFactorMax,
            "_withdrawFeeFactor too high"
        );
        withdrawFeeFactor = _withdrawFeeFactor;

        require(_buyBackRate <= buyBackRateUL, "_buyBackRate too high");
        buyBackRate = _buyBackRate;

        require(
            _slippageFactor <= slippageFactorUL,
            "_slippageFactor too high"
        );
        slippageFactor = _slippageFactor;

        emit SetSettings(
            _withdrawFeeFactor,
            _buyBackRate,
            _slippageFactor
        );
    }
```

Listing 3.5: Example Privileged Operations in `AVaultBase`

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 transfers the ownership to a timelock contract named `RewardsDistributorTimelock`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Avault` protocol, which is a yield aggregator platform that provides `aLP/aToken` to `DeFi` users with automated compounding yields at empirically optimal intervals while pooling gas fees through smart contracts and best yield optimization strategies. The current code base is clearly organized and 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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

[6] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[9] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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