SMART CONTRACT AUDIT REPORT

for

Adamantium

Prepared By: Patrick Lou

PeckShield

April 8, 2022

Document Properties

Client	Adamantium
Title	Smart Contract Audit Report
Target	Adamantium
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	April 8, 2022	Xuxian Jiang	Final Release
1.0-rc	April 7, 2022	Xuxian Jiang	Release Candidate

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 source code of the `Adamantium` 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 is well engineered without security-related issues. due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Adamantium

The `Adamantium` protocol provides a decentralized asset which rewards users with a fixed compound interest model through use of an `Auto Staking Protocol (ASP)`. It also performs cross chain farming with the `DEF`, a decentralized equity fund. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of the audited protocol

Item	Description
Name	Adamantium
Website	https://linktr.ee/AdamantiumVip
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 8, 2022

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

- https://github.com/DeflixFinance/adamantium-contracts.git (b44e460)

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

- https://github.com/DeflixFinance/adamantium-contracts.git (TBD)

1.2 About PeckShield

PeckShield Inc. [10] 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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

- 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) [8], 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 implementation of the `Adamantium` 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	1	■
Low	3	■ ■ ■
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 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 and 3 low-severity vulnerabilities.

Table 2.1: Key Audit Findings of Adamantium Protocol

ID	Severity	Title	Category	Status
PVE-001	Low	Proper _isFeeExempt Configuration	Business Logic	Resolved
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-003	Medium	Trust on Admin Keys	Security Features	Mitigated
PVE-004	Low	SubOptimal Swaps For Liquidity Addition	Coding Practices	Resolved

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 _isFeeExempt Configuration

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Adamantium`
- 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 `Adamantium` protocol is no exception. Specifically, it has defined a number of protocol-wide risk parameters, such as `initialDistributionFinished` and `isFeeExempt`. In the following, we show the corresponding routines that allow for their changes.

```
    function setFeeExempt(address _addr, bool _value) external onlyOwner {
        require(isFeeExempt[_addr] != _value, "Not changed");
        isFeeExempt[_addr] = _value;

        emit SetFeeExempt(_addr, _value);
    }

    function setSwapBackSettings(
        bool _enabled,
        uint256 _num,
        uint256 _denom
    ) external onlyOwner {
        swapEnabled = _enabled;
        _gonSwapThreshold = _total_gons.div(_denom).mul(_num);
        emit SetSwapBackSettings(_enabled, _num, _denom);
    }

    function setTargetLiquidity(uint256 target, uint256 accuracy)
        external
        onlyOwner
    {
```

```
        _targetLiquidity = target;
        _targetLiquidityDenominator = accuracy;
        emit SetTargetLiquidity(target, accuracy);
    }

    function setFeeReceivers(
        address _treasuryReceiver,
        address _riskFreeValueReceiver
    ) external onlyOwner {
        require(_treasuryReceiver != address(0), "_treasuryReceiver not set");
        require(
            _riskFreeValueReceiver != address(0),
            "_riskFreeValueReceiver not set"
        );
        treasuryReceiver = _treasuryReceiver;
        riskFreeValueReceiver = _riskFreeValueReceiver;
        emit SetFeeReceivers(_treasuryReceiver, _riskFreeValueReceiver);
    }
```

Listing 3.1: `Adamantium::setFeeExempt()/setSwapBackSettings()/setTargetLiquidity()/setFeeReceivers()`

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 `setFeeReceivers()` can be improved. In particular, when the new `treasuryReceiver` and `riskFreeValueReceiver` are used, there is a need to add them into the fee-exemption list (by marking the associated `_isFeeExempt` entries). By doing so, we avoid charging the fee in the fee collection for `treasuryReceiver` and `riskFreeValueReceiver` and maintain the consistency with the `constructor()` logic.

Recommendation Properly add the new `treasuryReceiver` and `riskFreeValueReceiver` in the fee-exemption list.

Status This issue has been fixed in the following commit: `9b22a0f`.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Adamantium`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [2]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.2: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `rescueToken()` routine in the `Adamantium` contract. If the `USDT` token is supported as `_tokenAddress`, the unsafe version of `IERC20(_tokenAddress).transfer(msg.sender, balance)` (line 553) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
546     function rescueToken(address _tokenAddress)
547         external
548         onlyOwner
549         returns (bool success)
```

```
    {
        require(_tokenAddress != address(this), "Can't withdraw ADM");
        uint256 balance = IERC20(_tokenAddress).balanceOf(address(this));
        return IERC20(_tokenAddress).transfer(msg.sender, balance);
    }
```

Listing 3.3: `Adamantium::rescueToken()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status This issue has been fixed in the following commit: `9b22a0f`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Adamantium`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In the `Adamantium` protocol, there are certain privileged accounts, i.e., `owner`. When examining the related contracts, we notice an inherent trust on these privileged accounts. For example, 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 . In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function setInitialDistributionFinished() external onlyOwner {
        initialDistributionFinished = true;
    }

    function setFeeExempt(address _addr, bool _value) external onlyOwner {
        require(isFeeExempt[_addr] != _value, "Not changed");
        isFeeExempt[_addr] = _value;

        emit SetFeeExempt(_addr, _value);
    }

    function setSwapBackSettings(
        bool _enabled,
        uint256 _num,
        uint256 _denom
    ) external onlyOwner {
```

```
        swapEnabled = _enabled;
        _gonSwapThreshold = _total_gons.div(_denom).mul(_num);
        emit SetSwapBackSettings(_enabled, _num, _denom);
    }
```

Listing 3.4: Example Privileged Operations in `Adamantium`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the `owner` 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 Make the list of extra privileges granted to `owner` explicit to the protocol users.

Status This issue has been confirmed and further mitigated by removing or restricting high-privileged and risky functions (e.g., minting and unlimited fee setting).

3.4 SubOptimal Swaps For Liquidity Addition

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Adamantium`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

As mentioned earlier, the `Adamantium` protocol is designed as a decentralized asset which rewards users with a fixed compound interest model through use of an `Auto Staking Protocol (ASP)`. Accordingly, there is a constant need of swapping one token to another. In the following, we examine related swap routines that are designed to assist the token swapping.

To elaborate, we show below a helper routine named `_doSwapBack()`. The routine is used to convert half `liquidityFee` to `BNB` and then add them as liquidity (`ADM_WBNB`) before sending them to the fee manager. It comes to our attention that the current approach converts half assets to `BNB` and then sends the another half with the converted `BNB` as liquidity, which may result in a small amount of `BNB` unspent in the current contract. In other words, the current conversion approach is not optimal.

```
    function _doSwapBack() private {
        uint256 dynamicLiquidityFee = isOverLiquified(
            _targetLiquidity,
            _targetLiquidityDenominator
        )
```

```
            ? 0
            : liquidityFee;
        uint256 contractTokenBalance = _gonBalances[address(this)].div(
            _gonsPerFragment
        );
        uint256 amountToLiquify = contractTokenBalance
            .mul(dynamicLiquidityFee)
            .div(totalFee)
            .div(2);
        uint256 amountToSwap = contractTokenBalance.sub(amountToLiquify);

        address[] memory path = new address[](2);
        path[0] = address(this);
        path[1] = router.WETH();

        uint256 balanceBefore = address(this).balance;

        router.swapExactTokensForETHSupportingFeeOnTransferTokens(
            amountToSwap,
            0,
            path,
            address(this),
            block.timestamp
        );

        uint256 amountETH = address(this).balance.sub(balanceBefore);

        uint256 totalETHFee = totalFee.sub(dynamicLiquidityFee.div(2));

        uint256 amountETHLiquidity = amountETH
            .mul(dynamicLiquidityFee)
            .div(totalETHFee)
            .div(2);
        uint256 amountETHRiskFreeValue = amountETH.mul(riskFreeValueFee).div(
            totalETHFee
        );
        uint256 amountETHTreasury = amountETH.mul(treasuryFee).div(totalETHFee);

        (bool success, ) = payable(treasuryReceiver).call{
            value: amountETHTreasury,
            gas: 30000
        }("");
        (success, ) = payable(riskFreeValueReceiver).call{
            value: amountETHRiskFreeValue,
            gas: 30000
        }("");

        success = false;

        if (amountToLiquify > 0) {
            router.addLiquidityETH{value: amountETHLiquidity}(
                address(this),
```

```
                amountToLiquify,
                0,
                0,
                DEAD,
                block.timestamp
            );
        }
    }
```

Listing 3.5: `Adamantium::_doSwapBack()`

Moreover, the above conversion does not specify any slippage restriction, which may be easily exploited in a possible sandwich or MEV attack for reduced return.

Recommendation Perform an optimal allocation of assets between two tokens for matched liquidity addition. Also add necessary slippage control to avoid unnecessary loss of swaps.

Status The issue has been acknowledged as a design choice.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Adamantium` protocol, which provides a decentralized asset and rewards users with a fixed compound interest model through use of an `Auto Staking Protocol (ASP)`. The current code base is well organized and 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.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-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: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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

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

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

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