SMART CONTRACT AUDIT REPORT

for

Algem Adapter

Prepared By: Xiaomi Huang

PeckShield

October 8, 2022

Document Properties

Client	Algem
Title	Smart Contract Audit Report
Target	Algem Adapter
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 8, 2022	Jing Wang	Final Release
1.0-rc	September 02, 2022	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 Algem Adapter design document and related smart contract source code, 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 Algem Adapter

The Algem Adapter have a serial of intermediary contracts between the user and the liquidity-providing platform. The adapter contracts will receive tokens and proxy the actions to the liquidity-providing media. Once locked in the adapter contract, the balance cannot be changed unless the liquidity is removed. Users can transfer tokens into the Adapter contracts through a single transaction. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Algem Adapter

Item	Description
Issuer	Algem
Website	https://www.algem.io/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 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/DippyArtu/algem/blob/main/packages/hardhat/contracts/SiriusAdapter.sol (06dd93f)

- https://github.com/azhlbn/for-audit/blob/main/contracts/ArthswapAdapter.sol (295ce5b8)

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

- https://github.com/azhlbn/for-audit/blob/main/contracts/SiriusAdapter.sol (b90df4e4)
- https://github.com/azhlbn/for-audit/blob/main/contracts/ArthswapAdapter.sol (b90df4e4)

1.2 About PeckShield

PeckShield Inc. [9] 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 the OWASP Risk Rating Methodology [8]:

- 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.

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Algem 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

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 checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Algem Adapter` 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	1	■
Low	2	■ ■
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 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 high-severity vulnerability, 1 medium-severity vulnerability, and 2 low-severity vulnerabilities.

Table 2.1: Key Algem Adapter Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Proper rewardDebt Accounting in SiriusAdapter	Business Logic	Fixed
PVE-002	Low	Accommodation of Possible Non-ERC20-Compliance	Business Logic	Fixed
PVE-003	Low	Possible Bypass of isContract() Sanity Check	Coding Practices	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	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 Proper rewardDebt Accounting in SiriusAdapter

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `SiriusAdapter`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, the `SiriusAdapter` contract is designed to allow users to stake `$nASTR` tokens and receive rewards. While reviewing this contract, we notice that the internal rewards-related accounting logic needs to be improved.

To elaborate, we show below the `depositLP()` routine, which is designed to allow users to deposit `LP` tokens and earn accrued rewards. This function updates the `rewardDebt` state to keep track of the current amount of rewards, which has already been distributed to user. However, it comes to our attention that the `rewardDebt` state is updated to `_amount * accumulatedRewardsPerShare`, which means the previous accumulated rewards delivered to the user have been cleaned up. A bad user could first deposit a large amount of `LP` tokens and then deposit `1 Wei` to drain all the rewards from the contract.

```
    function depositLP(uint256 _amount) public update {
        require(lpBalances[msg.sender] >= _amount, "Not enough LP tokens");
        require(_amount > 0, "Shoud be greater than zero");
        require(!(msg.sender.isContract()), "Allows only for external owned accounts");

        lpBalances[msg.sender] -= _amount;

        uint256 beforeGauge = gauge.balanceOf(address(this));
        farm.deposit(_amount, address(this), false);
        uint256 afterGauge = gauge.balanceOf(address(this));
        uint256 receivedGauge = afterGauge - beforeGauge;

```

```
        gaugeBalances[msg.sender] += receivedGauge;
        rewardDebt[msg.sender] = _amount * accumulatedRewardsPerShare /
            REWARDS_PRECISION;
        emit DepositLP(msg.sender, _amount);
    }
```

Listing 3.1: `SiriusAdapter::depositLP()`

Note other routines, including `addGauge()` and `withdrawLP()`, share the same issue.

Recommendation Properly update the `rewardDebt` state using the current balance of the user have.

Status This issue has been fixed in the commit: b90df4e4.

3.2 Accommodation of Possible Non-ERC20-Compliance

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SiriusAdapter`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

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 this section, we examine the `transferFrom()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. Specifically, the `transferFrom()` routine does not have a return value defined and implemented. However, the `IERC20` interface has defined the `transferFrom()` interface with a `bool` return value. As a result, the call to `transferFrom()` may expect a return value. With the lack of return value of `USDT`'s `transferFrom()`, the call will be unfortunately reverted.

```
    function transferFrom(address _from, address _to, uint _value) public
        onlyPayloadSize(3 * 32) {
        var _allowance = allowed[_from][msg.sender];

        // Check is not needed because sub(_allowance, _value) will already throw if
            this condition is not met
        // if (_value > _allowance) throw;

        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
```

```
        }
        if (_allowance < MAX_UINT) {
            allowed[_from][msg.sender] = _allowance.sub(_value);
        }
        uint sendAmount = _value.sub(fee);
        balances[_from] = balances[_from].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(_from, owner, fee);
        }
        Transfer(_from, _to, sendAmount);
    }
```

Listing 3.2: USDT Token **Contract**

Because of that, a normal call to `transferFrom()` is suggested to use the safe version, i.e., `safeTransferFrom()`, 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()/transfer()` as well, i.e., `safeApprove()/safeTransfer()`.

In current implementation, if we examine the `SiriusAdapter::addLiquidity()` routine that is designed to add liquidity to the pool with the given amounts of tokens. To accommodate the specific idiosyncrasy, there is a need to use `safeTransferFrom()`, instead of `transferFrom()` (line 128).

```
123     function addLiquidity(uint256[] calldata _amounts, bool _autoStake) external payable
            update {
124         require(!(msg.sender.isContract()), "Allows only for external owned accounts");
125         require(msg.value == _amounts[0], "Value need to be equal to amount of ASTR
                tokens");
126         require(_amounts[0] > 0 && _amounts[1] > 0, "Amounts of tokens should be greater
                 than zero");

128         require(nToken.transferFrom(msg.sender, address(this), _amounts[1]), "Error
                while nASTR transfer");

130         uint256 lpAmount = pool.addLiquidity{value: msg.value}(_amounts, 0, block.
                timestamp + 1200);
131         lpBalances[msg.sender] += lpAmount;

133         if (_autoStake) {
134             depositLP(lpAmount);
135         }
136         emit AddLiquidity(msg.sender, _amounts, _autoStake, lpAmount);
137     }
```

Listing 3.3: `SiriusAdapter::addLiquidity()`

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

Status This issue has been fixed in the commit: b90df4e4.

3.3 Possible Bypass of isContract() Sanity Check

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `SiriusAdapter`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

The `SiriusAdapter` contract allows users to stake `LP` tokens to receive rewards. In order to reduce the risks of the contract being attacked by malicious users, the key functions in the contract use `isContract()` to prevent contracts calling them directly.

```
    function removeLiquidity(uint256 _amount) public update {
        require(_amount > 0, "Should be greater than zero");
        require(lpBalances[msg.sender] >= _amount, "Not enough LP");
        require(!(msg.sender.isContract()), "Allows only for external owned accounts");
        ...
    }
```

Listing 3.4: `SiriusAdapter::removeLiquidity()`

The function `isContract()` determines whether the caller is a contract by checking the `address.code.length` of the caller (line 41). However, a contract does not have code available during its construction. So, if a contract makes calls to other contracts inside the `constructor()`, the `address.code.length` would be 0, which allows the caller to bypass the `isContract()` check.

```
36    function isContract(address account) internal view returns (bool) {
37        // This method relies on extcodesize/address.code.length, which returns 0
38        // for contracts in construction, since the code is only stored at the end
39        // of the constructor execution.
40
41        return account.code.length > 0;
42    }
```

Listing 3.5: Address.sol

Recommendation Ensure the `msg.sender` is the same as `tx.origin`.

```
    modifier notAllowContract() {
```

```
        require(!msg.sender.isContract() && tx.origin == msg.sender, "Allows only for
            external owned accounts");
        _;
    }
```

Listing 3.6: SiriusAdapter.sol

Status This issue has been fixed in the commit: b90df4e4.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: High
- Target: `SiriusAdapter`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Algem Adapter` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the protocol-wide operations (e.g., funds withdrawal and parameter configuration). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged `owner` account and its related privileged accesses in current contract.

To elaborate, we show the `withdraw()` routine from the `SiriusAdapter` contract. This function allows the `owner` account withdraw all native tokens from the contract.

```
    // @notice It is not supposed that funds will be accumulated on the contract
    //         This reserve function is needed to withdraw stucked funds
    function withdraw() external onlyOwner {
        payable(msg.sender).transfer(address(this).balance);
    }
```

Listing 3.7: SiriusAdapter::withdraw()

We understand the need of the privileged functions for contract maintenance, but it is worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

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 confirmed. The team clarifies they plan on using an `EOA` to start, and eventually migrating ownership of sensitive contracts to multi-sig in the future and switch to `DAO`-like governance contract.

4 Conclusion

In this audit, we have analyzed the Algem Adapter design and implementation. Algem Adapter is an intermediary contract between the user and the liquidity-providing platform. The current code base is well structured and neatly organized. 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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 CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

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

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

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