SMART CONTRACT AUDIT REPORT

for

OnePiece

"OnePiece" has officially changed its name to "CIAN" in May

Prepared By: Patrick Lou

PeckShield

April 25, 2022

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	April 25, 2022	Luck Hu	Final Release
1.0-rc	April 11, 2022	Luck Hu	Release Candidate

Contact

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

Name	Patrick Lou
Phone	+86 156 0639 2692
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the OnePiece 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 contract can be further improved due to the presence of several issues related to business logic, security or performance. This document outlines our audit results.

1.1 About OnePiece

OnePiece is an automation platform contributed and utilized by users to improve onchain efficiency and capital utilization. It aims to be a DeFi operating system that redefines the way you perform a DeFi task. It substitutes the intricate, time-consuming manual operations with simple task definition of few clicks. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of the OnePiece

Item	Description
Name	OnePiece
Website	https://onepiece.ai//
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 25, 2022

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

- https://github.com/onepiece-ai/onepiece-protocol-audit.git (037ce54)

And this is the commit ID after all fixes for the issues found in the audit have been checked in: (Note PVE005 is partially fixed by this commit.)

- https://github.com/onepiece-ai/onepiece-protocol-audit.git (88241c5)

1.2 About PeckShield

PeckShield Inc. [13] 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 [12]:

- 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.

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

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 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) [11], 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 `OnePiece` design and 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 logics, 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	2	■ ■
Low	4	■ ■ ■ ■
Informational	0	
Total	7	

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 some 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, the audited protocol is 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, 2 medium-severity vulnerabilities, and 4 low-severity vulnerabilities.

Table 2.1: Key OnePiece Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Improper Authorization Checks in ControllerLink	Security Features	Fixed
PVE-002	Medium	Lack Of Timelocked Tx Execution in TimeLock	Business Logic	Fixed
PVE-003	Low	Improved Logic of ERC2612Verifier::permit()	Coding Practices	Fixed
PVE-004	Low	Consistent Deadline Handling Between ERC2612Verifier::permit() & isTxPermitted()	Coding Practices	Fixed
PVE-005	Low	Possible Double Initialization From Initializer Reentrancy	Coding Practices	Fixed
PVE-006	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practice	Fixed
PVE-007	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed

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 Improper Authorization Checks in ControllerLink

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `ControllerLink`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

In the `OnePiece` protocol, there is a `ControllerLink` contract to maintain a user `database`. With the database, users could be added via the `addAuth()` routine, and removed via the `removeAuth()` routine. While examining these two routines, we notice the existence of improper authorization checks that need to be corrected.

To elaborate, we show below the code snippets of these two routines. It comes to our attention that there is no access control restriction enforced on the `addAuth()` routine, which allows anyone to invoke it. In addition, the authorization check (line 68) of the `removeAuth()` routine could be easily bypassed by a malicious user since the validation check depends on the untrusted input! Specifically, the calling contract (`msg.sender`) can simply return `false` in its crafted `isAuth()` implementation to bypass the validation check.

```
56    function addAuth(address _owner, address _account) external {
57        accounts++;
58        accountID[_account] = accounts;
59        accountAddr[accounts] = _account;
60        addAccount(_owner, accountID[_account]);
61        addUser(_owner, accountID[_account]);

63        emit NewAccount(_owner, _account);
64    }

66    function removeAuth(address _owner, address _account) external {
67        require(accountID[_account] != 0, "not-account");
```

```
        require(!AccountInterface(msg.sender).isAuth(_owner), "already-owner");
        removeAccount(_owner, accountID[_account]);
        removeUser(_owner, accountID[_account]);
    }
```

Listing 3.1: ControllerLink.sol

Our assessment shows that the above improper authorization check will make the `addAuth()/removeAuth()` routines open to public. Therefore, it is suggested to add necessary access controls to better protect users and their assets.

Recommendation Add the necessary access control authorization to the `addAuth()/removeAuth()` routines.

Result The issue has been fixed by this commit: `e5c88a9`.

3.2 Lack Of Timelocked Tx Execution in TimeLock

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `Timelock`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The `OnePiece` protocol has an `AdapterManager` contract to mediate and manage all the `adapters` used in the protocol. Our analysis shows that the registration of new `adapters` and un-registration of existing ones are controlled by the `Timelock` contract, which queues and executes proposals that survive a governance vote. The `Timelock` contract has a `delay period` (12 hours-30 days), which defines the lock time of the queued transaction before it can be executed. And the `Timelock` contract also defines a `GRACE_PERIOD` (14 days) which gives the last time before which the transaction must be executed after it has passed the time-locked period. While examining the time lock logic, we notice the time lock is currently disabled (lines 127 – 130 and 179 – 186). Without the time lock enforcement, any transaction could be queued and executed instantly.

```
116     function queueTransaction(
117         address target,
118         uint256 value,
119         string memory signature,
120         bytes memory data,
121         uint256 eta
122     ) public returns (bytes32) {
123         require(
```

```
            msg.sender == admin,
            "Timelock::queueTransaction: Call must come from admin."
        );
        // require(
        //     eta >= getBlockTimestamp() + delay,
        //     "Timelock::queueTransaction: Estimated execution block must satisfy delay."
        // );
        ...
        return txHash;
    }
```

Listing 3.2: `Timelock::queueTransaction()`

```
    function executeTransaction(
        address target,
        uint256 value,
        string memory signature,
        bytes memory data,
        uint256 eta
    ) public payable returns (bytes memory) {
        require(
            msg.sender == admin,
            "Timelock::executeTransaction: Call must come from admin."
        );

        bytes32 txHash = keccak256(
            abi.encode(target, value, signature, data, eta)
        );
        require(
            queuedTransactions[txHash],
            "Timelock::executeTransaction: Transaction hasn't been queued."
        );
        // require(
        //     getBlockTimestamp() >= eta,
        //     "Timelock::executeTransaction: Transaction hasn't surpassed time lock."
        // );
        // require(
        //     getBlockTimestamp() <= eta + GRACE_PERIOD,
        //     "Timelock::executeTransaction: Transaction is stale."
        // );

        queuedTransactions[txHash] = false;
        ...
        return returnData;
    }
```

Listing 3.3: `Timelock::executeTransaction()`

Recommendation Enforce the above-mentioned time lock logic in the `Timelock` contract.

Status The issue has been fixed by this commit: `a6108b8`.

3.3 Improved Logic of ERC2612Verifier::permit()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `ERC2612Verifier`
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [4]

Description

To facilitate the user interaction, the `OnePiece` protocol has an `ERC2612Verifier` contract to support the `EIP2612`-compliant functionality. In particular, the `permit()` function is introduced to simplify the call forwarding process.

To elaborate, we show below this `permit()` routine in the `ERC2612Verifier` contract. This routine ensures that the owner of the given `account` is indeed the one who signs the approve request. Note that the internal implementation makes use of the `ecrecover()` precompile for validation (line 89). It comes to our attention that the precompile-based validation needs to properly ensure the signer, i.e., `OwnableUpgradeable(account).owner()`, is not equal to `address(0)`. Because the `ecrecover()` will return `address(0)` for any failure. If the owner of the given `account` renounces the ownership, then the owner becomes `address(0)`. As a result, anybody could approve a function call forwarded to the given `account`.

```
63     function permit(
64         address account,
65         address operator,
66         bytes32 approvalType,
67         uint256 deadline,
68         uint8 v,
69         bytes32 r,
70         bytes32 s
71     ) external {
72         require(deadline >= block.timestamp, "Permit: EXPIRED");
73         bytes32 digest = keccak256(
74             abi.encodePacked(
75                 "\x19\x01",
76                 DOMAIN_SEPARATOR,
77                 keccak256(
78                     abi.encode(
79                         PERMIT_TYPEHASH,
80                         account,
81                         operator,
82                         approvalType,
83                         nonces[account]++,
84                         deadline
85                     )
86                 )
```

```
            )
        );
        address recoveredAddress = ecrecover(digest, v, r, s);
        require(
            OwnableUpgradeable(account).owner() == recoveredAddress,
            "not the owner of the address"
        );
        approvals_deadline[account][operator] = deadline;
        emit OperatorUpdate(account, operator);
    }
```

Listing 3.4: `ERC2612Verifier::permit()`

Recommendation Strengthen the `permit()` routine to ensure the `recoveredAddress` is not equal to `address(0)`.

Status The issue has been fixed by this commit: `e85a2a6`.

3.4 Consistent Deadline Handling Between ERC2612Verifier::permit() & isTxPermitted()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ERC2612Verifier`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1041 [1]

Description

As mentioned earlier, the `ERC2612Verifier` contract has the `permit()` function to simplify the call forwarding process. Specifically, it approves a transaction on the give `account` after validating the signer of the request. The approval is controlled by the `deadline` argument, which defines the valid time of the approval. After the `deadline` expires, the approval becomes stale. While examining the `deadline` validation, we observe an inconsistency between the `permit()` routine and the `isTxPermitted()` routine.

To elaborate, we show below the code snippet from the `ERC2612Verifier` contract. Specially, in the `permit()` routine, the valid time of the approval includes the `deadline` (line 72). While in the `isTxPermitted()` routine, the valid time of the approval does not include the `deadline` (line 103). It is suggested to keep them consistent.

```
63    function permit(
64        address account,
65        address operator,
```

```
        bytes32 approvalType,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external {
        require(deadline >= block.timestamp, "Permit: EXPIRED");
        bytes32 digest = keccak256(
            abi.encodePacked(
                "\x19\x01",
                DOMAIN_SEPARATOR,
                keccak256(
                    abi.encode(
                        PERMIT_TYPEHASH,
                        account,
                        operator,
                        approvalType,
                        nonces[account]++,
                        deadline
                    )
                )
            )
        );
        address recoveredAddress = ecrecover(digest, v, r, s);
        require(
            OwnableUpgradeable(account).owner() == recoveredAddress,
            "not the owner of the address"
        );
        approvals_deadline[account][operator] = deadline;
        emit OperatorUpdate(account, operator);
    }

    function isTxPermitted(
        address account,
        address operator,
        uint256
    ) external view override returns (uint256) {
        if (approvals_deadline[account][operator] > block.timestamp) {
            return 1;
        }
        return 0;
    }
```

Listing 3.5: ERC2612Verifier.sol

Recommendation Revise the above mentioned routines to make the `deadline` check of the approval consistent.

Status The issue has been fixed by this commit: `e85a2a6`.

3.5 Possible Double Initialization From Initializer Reentrancy

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Multiple Contracts
- Category: Time and State [10]
- CWE subcategory: CWE-682 [5]

Description

The `OnePiece` protocol supports flexible contract initialization, so that the initialization task does not need to be performed inside the constructor at deployment. This feature is enabled by introducing the `initializer()` modifier that protects an initializer function from being invoked twice. It becomes known that the popular `OpenZepplin` reference implementation has an issue that makes it possible to re-enter `initializer()`-protected functions. In particular, for this to happen, one call may need to be a nested-call of the other, or both calls have to be subcalls of a common `initializer()`-protected function. You can find more in detail about this issue from: `#3006`.

The reentrancy can be dangerous as the initialization is not part of the proxy construction, and it becomes possible by executing an external call to an untrusted address. As part of the fix, there is a need to forbid `initializer()`-protected functions to be nested when the contract is already constructed.

To elaborate, we show below the current `initializer()` implementation as well as the fixed implementation.

```
    modifier initializer() {
        require(
            _initializing || !_initialized,
            "Initializable: contract is already initialized"
        );

        bool isTopLevelCall = !_initializing;
        if (isTopLevelCall) {
            _initializing = true;
            _initialized = true;
        }

        _;

        if (isTopLevelCall) {
            _initializing = false;
        }
    }
```

Listing 3.6: `Initializable::initializer()`

```
    modifier initializer() {
        require(_initializing? _isConstructor() : !_initialized, "Initializable:
            contract is already initialized");

        bool isTopLevelCall = !_initializing;
        if (isTopLevelCall) {
            _initializing = true;
            _initialized = true;
        }

        _;

        if (isTopLevelCall) {
            _initializing = false;
        }
    }
```

Listing 3.7: Revised `Initializable::initializer()`

Recommendation Enforce the `initializer()` modifier to prevent it from being re-entered.

Status The issue has been partially fixed by this commit: `3494535`.

3.6 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- 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 this section, we examine the `approve()` 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. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
```

```
     * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
     * @param _spender The address which will spend the funds.
     * @param _value The amount of tokens to be spent.
     */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.8: USDT Token **Contract**

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

More importantly, the `approve()` function of some token may return false while not revert on failure. Accordingly, the call to `approve()` is expected to check the return value. If it returns false, the call to `approve()` shall be failed.

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`, 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. To use this library you can add a using SafeERC20 for IERC20. Similarly, there is a safe version of `transfer()/transferFrom()` as well, i.e., `safeTransfer()/safeTransferFrom()`.

In the following, we show the `approve()` routine in the `ControllerLib` contract. If the `approve()` of the given `token` does not revert on failure, the unsafe version of `IERC20(token).approve(to, amount)` (line 244) need to check the return value while not assuming the `approve()` will revert internally.

```
239    function approve(
240        address token,
241        address to,
242        uint256 amount
243    ) external onlyPermit {
244        IERC20(token).approve(to, amount);
245    }
```

Listing 3.9: ControllerLib :: approve()

Note same issue exists in the `ControllerLib` contract and the `TraderJoeAdapter` contract.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`. And there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Status The issue has been fixed by this commit: `13c8dd6`.

3.7 Trust Issue Of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

In the `OnePiece` protocol, there exist certain privileged accounts that play critical roles in governing and regulating the protocol-wide operations. In the following, we examine these privileged accounts and their related privileged accesses in current contracts.

Firstly, the privileged functions in the `ControllerLib` contract allow for the the `owner` to withdraw all the tokens from the contract. And the `owner/_automation` are privileged to approve tokens transfer from current contract to other amounts, etc.

```
    function withdrawAsset(
        address _token,
        address _recipient,
        uint256 _amount
    ) external onlyOwner {
        if (_recipient != owner()) {
            require(advancedOptionEnable, "Not allowed!");
        }
        bool isEth = _token == avaxAddr;
        if (isEth) {
            uint256 _balance = address(this).balance;
            require(_balance >= _amount, "not enough AVAX balance");
            safeTransferAVAX(_recipient, _amount);
        } else {
            uint256 _balance = IERC20(_token).balanceOf(address(this));
            require(_balance >= _amount, "not enough token balance");
            IERC20(_token).transfer(_recipient, _amount);
        }
    }
```

Listing 3.10: `ControllerLib::withdrawAsset()`

```
function approve(
    address token,
    address to,
    uint256 amount
) external onlyPermit {
    IERC20(token).approve(to, amount);
}
```

Listing 3.11: `ControllerLib::approve()`

Secondly, the privileged function in the `CallProxyLib` contract allows for the the `owner` to set the whitelist of the loan providers.

```
    function setFlashLoanWhiteList(address protocol, bool val)
        external
        onlyOwner
    {
        flashLoanWhiteList[protocol] = val;
    }
```

Listing 3.12: `CallProxyLib::setFlashLoanWhiteList()`

Lastly, the privileged functions in the `AdapterManager` contract allow for the the `owner` to set the `partners` who can pause the contract. The `owner` is also privileged to pause/unpause the contract.

```
    function setPauseWhiteList(address partner, bool val) external onlyOwner {
        if (val == false) {
            require(suspendPermissions[partner], "No change.");
        } else {
            require(!suspendPermissions[partner], "No change.");
        }
        suspendPermissions[partner] = val;
    }

    function setPause(bool val) external {
        if (val == true) {
            require(
                suspendPermissions[msg.sender]  msg.sender == owner(),
                "verification failed."
            );
        } else {
            require(msg.sender == owner(), "verification failed.");
        }
        _paused = val;
        if (_paused) {
            emit Paused();
        } else {
            emit Unpaused();
        }
    }
```

Listing 3.13: `AdapterManager.sol`

There are also some other privileged functions not listed above. And 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 Make the list of extra privileges granted to `owner/_automation`, etc. explicit to `OnePiece` protocol users.

Status This issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the `OnePiece` design and implementation. The protocol is designed to be an automation platform for users to improve onchain efficiency and capital utilization. During the audit, we notice that the current code base is well organized. and those identified issues are promptly confirmed and fixed.

Moreover, 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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

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

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

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

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

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

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

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