SMART CONTRACT AUDIT REPORT

for

Acala Euphrates (v2)

Prepared By: Xiaomi Huang

PeckShield

January 5, 2024

Document Properties

Client	Acala
Title	Smart Contract Audit Report
Target	Acala
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 5, 2024	Xuxian Jiang	Final Release
1.0-rc	January 3, 2024	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the Acala Euphrates (v2) 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 Acala Euphrates

Acala is the decentralized finance network and liquidity hub of Polkadot. It is a layer-1 smart contract platform that is scalable, Ethereum-compatible, and optimized for DeFi with built-in liquidity and ready-made financial applications. The audited Euphrates (v2) provides the staking support as well as the related converters among DOT, LCDOT, TDOT and WTDOT by externally interacting with system contracts (LiquidCrowdloan, Homa, and StableAssets). The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Acala

Item	Description
Name	Acala
Website	https://acala.network/
Type	Polkadot
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 5, 2024

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

- https://github.com/AcalaNetwork/Euphrates.git (0a76674)

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

- https://github.com/AcalaNetwork/Euphrates.git (b9d2682)

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 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.

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 accordingly classified into four categories, 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) [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 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 `Acala Euphrates (v2)` implementations. 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	2	
Informational	0	
Total	3	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities 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 2 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved _convert() Logic in DOT2WTDOTConvertor	Time and State	Resolved
PVE-002	Low	Inconsistent LST Conversion in UpgradeableStakingLST	Coding Practices	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Improved _convert() Logic in DOT2WTDOTConvertor

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Time and State [6]
- CWE subcategory: CWE-682 [3]

Description

As mentioned earlier, the Euphrates (v2) protocol provides built-in converters among DOT, LCDOT, TDOT and WTDOT by externally interacting with system contracts such as LiquidCrowdloan, Homa, and StableAssets. Our analysis shows certain conversion does not enforce meaningful slippage control.

```
    function _convert(
        uint256 inputAmount,
        address receiver
    ) internal returns (uint256 outputAmount) {
        require(inputAmount != 0, "DOT2WTDOTConvertor: invalid input amount");
        IERC20(dot).safeTransferFrom(msg.sender, address(this), inputAmount);

        // params for tDOT pool of StableAsset on Acala:
        // tDOT pool id: 0
        // assets length: 2
        // asset index of DOT: 0
        // asset index of LDOT: 1
        // here deadcode these params
        (bool valid, address[] memory assets) = IStableAsset(stableAsset)
            .getStableAssetPoolTokens(0);
        require(
            valid && assets[0] == dot,
            "DOT2WTDOTConvertor: invalid stable asset pool"
        );
        uint256[] memory paramAmounts = new uint256[](2);

        if (inputAmount.div(2) >= HOMA_MINT_THRESHOLD) {
```

```
118             uint256 beforeLdotAmount = IERC20(ldot).balanceOf(address(this));
119             bool suc = IHoma(homa).mint(inputAmount.div(2));
120             require(suc, "DOT2WTDOTConvertor: homa mint failed");
121             uint256 afterLdotAmount = IERC20(ldot).balanceOf(address(this));
122             uint256 ldotAmount = afterLdotAmount.sub(beforeLdotAmount);
123 
124             // convert LDOT amount to rebased LDOT amount as the param
125             // NOTE: the precision of Homa.getExchangeRate is 1e18
126             uint256 ldotParamAmount = ldotAmount
127                 .mul(IHoma(homa).getExchangeRate())
128                 .div(1e18);
129             paramAmounts[0] = inputAmount.sub(inputAmount.div(2));
130             paramAmounts[1] = ldotParamAmount;
131         } else {
132             paramAmounts[0] = inputAmount;
133             paramAmounts[1] = 0;
134         }
135 
136         uint256 beforeTdotAmount = IERC20(tdot).balanceOf(address(this));
137         bool success = IStableAsset(stableAsset).stableAssetMint(
138             0,
139             paramAmounts,
140             0
141         );
142         require(success, "DOT2WTDOTConvertor: stable-asset mint failed");
143         uint256 afterTdotAmount = IERC20(tdot).balanceOf(address(this));
144         uint256 tdotAmount = afterTdotAmount.sub(beforeTdotAmount);
145 
146         IERC20(tdot).safeApprove(wtdot, tdotAmount);
147         outputAmount = IWTDOT(wtdot).deposit(tdotAmount);
148 
149         require(outputAmount > 0, "DOT2WTDOTConvertor: zero output");
150         IERC20(wtdot).safeTransfer(receiver, outputAmount);
151     }
```

Listing 3.1: DOT2WTDOTConvertor::_convert()

To elaborate, we show above one example routine `_convert()` from the `DOT2WTDOTConvertor` contract. We notice the conversion is routed to an external `stableAsset` in order to swap given assets to another `TDOT`. And the swap operation does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of conversion.

In addition, the above routine can also be improved by additionally requiring `assets[1] == ldot` (line 112).

Recommendation Develop an effective mitigation (e.g., slippage control) to the above front-running attack to better protect the interests of staking users. Note the same issue is also applicable to other contracts, including `LCDOT2WTDOTConvertor` and `UpgradeableStakingLST`.

Status This issue has been addressed in the following commit: `b9d2682`.

3.2 Inconsistent LST Conversion in UpgradeableStakingLST

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `UpgradeableStakingLST`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The `Euphrates` protocol allows users to stake supported assets and get in return the respective share. While examining the staking logic, we notice the token conversion is not consistent with provided converters.

To elaborate, we show below the related `stake()` routine, which is used for participating users to stake the supported assets. We notice two supported `shareTypes` (`DOT` and `LCDOT`) as well as three `convertedShareTypes` (`LDOT`, `TDOT`, and `WTDOT`). However, `TDOT` is not supported as part of `convertedShareType` in `convertLSTPool()` function in the same contract.

```
    function stake(uint256 poolId, uint256 amount)
        public
        virtual
        override
        whenNotPaused
        poolOperationNotPaused(poolId, Operation.Stake)
        updateRewards(poolId, msg.sender)
        nonReentrant
        returns (bool)
    {
        IERC20 shareType = shareTypes(poolId);
        require(address(shareType) != address(0), "invalid pool");

        uint256 addedShare;
        ConvertInfo memory convertInfo = convertInfos(poolId);
        if (address(convertInfo.convertedShareType) != address(0)) {
            // if pool has converted, transfer the before share token to this firstly
            shareType.safeTransferFrom(msg.sender, address(this), amount);

            uint256 convertedAmount;
            if (address(shareType) == LCDOT && address(convertInfo.convertedShareType)
                == LDOT) {
                convertedAmount = _convertLCDOT2LDOT(amount);
            } else if (address(shareType) == LCDOT && address(convertInfo.
                convertedShareType) == TDOT) {
                convertedAmount = _convertLCDOT2TDOT(amount);
```

```
            } else if (address(shareType) == DOT && address(convertInfo.convertedShareType) == LDOT) {
                convertedAmount = _convertDOT2LDOT(amount);
            } else if (address(shareType) == DOT && address(convertInfo.convertedShareType) == TDOT) {
                convertedAmount = _convertDOT2TDOT(amount);
            } else if (address(shareType) == LCDOT && address(convertInfo.convertedShareType) == WTDOT) {
                uint256 tdotAmount = _convertLCDOT2TDOT(amount);
                convertedAmount = _convertTDOT2WTDOT(tdotAmount);
            } else if (address(shareType) == DOT && address(convertInfo.convertedShareType) == WTDOT) {
                uint256 tdotAmount = _convertDOT2TDOT(amount);
                convertedAmount = _convertTDOT2WTDOT(tdotAmount);
            } else {
                revert("unsupported converted share token");
            }

            // must convert the share amount according to the exchange rate of converted pool
            addedShare = convertedAmount.mul(1e18).div(convertInfo.convertedExchangeRate);
        } else ...

        require(addedShare > 0, "cannot stake 0");
        _totalShares[poolId] = _totalShares[poolId].add(addedShare);
        _shares[poolId][msg.sender] = _shares[poolId][msg.sender].add(addedShare);

        emit Stake(msg.sender, poolId, addedShare);

        return true;
    }
```

Listing 3.2: `UpgradeableStakingLST::stake()`

Recommendation Revise the above staking logic to be consistent with the conversion routine `convertLSTPool()`

Status The issue has been resolved as the team upgrades it in `UpgradeableStakingLSTV2`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Euphrates (v2)` 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., parameter configuration and pool adjustment). 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 contracts.

```
    function pause() external onlyOwner {
        _pause();
    }

    /// @notice Unpause the contract by Pausable.
    /// @dev Define the `onlyOwner` access.
    function unpause() external onlyOwner {
        _unpause();
    }

    /// @inheritdoc PoolOperationPausable
    /// @dev Override the inherited function to define `onlyOwner` and `whenNotPaused` access.
    function setPoolOperationPause(uint256 poolId, Operation operation, bool paused)
        public
        override
        onlyOwner
        whenNotPaused
    {
        super.setPoolOperationPause(poolId, operation, paused);
    }

    /// @inheritdoc Staking
    /// @dev Override the inherited function to define `onlyOwner` and `whenNotPaused` access.
    function addPool(IERC20 shareType) public override onlyOwner whenNotPaused {
        super.addPool(shareType);
    }
```

Listing 3.3: Example Privileged Operations in `UpgradeableStakingCommon`

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 mitigatd as the team clarifies the use of multisig to manage the admin keys.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Acala Euphrates (v2)` protocol, which provides the staking support as well as the related converters among `DOT`, `LCDOT`, `TDOT` and `WTDOT` by externally interacting with system contracts (`LiquidCrowdloan`, `Homa`, and `StableAssets`). The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.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: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.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.