



SMART CONTRACT AUDIT REPORT

for

Acala Euphrates



Prepared By: Xiaomi Huang

PeckShield
September 12, 2023

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	September 12, 2023	Xuxian Jiang	Final Release
1.0-rc	September 9, 2023	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	4
1.1	About Acala Euphrates	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Simplified depositRate() Logic in WrappedTDOT	11
3.2	Possible Costly WTDOT From Improper Initialization	12
3.3	Suggested Adherence Of Checks-Effects-Interactions Pattern	13
3.4	Trust Issue of Admin Keys	15
4	Conclusion	17
	References	18

1 | Introduction

Given the opportunity to review the design document and related source code of the `Acala Euphrates` 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` support is for `DOT` `LSD` farming that will deploy at `Acala EVM+`. The contract can convert assets (`LcDOT/DOT`) of the pool into `LSD` assets (`LDOT/tDOT`) by external call the system contracts (`LiquidCrowdloan`, `Homa`, and `StableAssets`) which bridge the pallet module of the `Acala` runtime. 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	September 12, 2023

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> (42ced0a)

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> (d261c81)

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	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

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 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) [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. 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 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	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 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

ID	Severity	Title	Category	Status
PVE-001	Low	Simplified depositRate() Logic in WrappedTDOT	Coding Practices	Resolved
PVE-002	Low	Possible Costly WTDOT From Improper Initialization	Time and State	Resolved
PVE-003	Low	Suggested Adherence of Checks-Effects-Interactions in Staking	Coding Practices	Resolved
PVE-004	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 Simplified depositRate() Logic in WrappedTDOT

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: WrappedTDOT
- Category: Coding Practices [7]
- CWE subcategory: CWE-1109 [1]

Description

The Euphrates protocol has a `WrappedTDOT` contract to wrap the TDOT tokens. The wrapping and unwrapping will have the respective `depositRate` and `withdrawRate`. While examining the `depositRate` calculation, we notice current implementation can be improved.

Specifically, we show below the code snippet from the `depositRate()` routine. This routine computes the rate from the deposited TDOT to the wrapped WTDOT. And the computation requires current total supply of WTDOT. And the used approach of making a low-level call `wtdotAmount = this.totalSupply()` (line 64) can be simplified as `wtdotAmount = totalSupply`. In addition, the returned `depositRate` under the condition of `wtdotAmount != 0 AND tdotAmount == 0` is infeasible.

```

62     function depositRate() public view returns (uint256) {
63         uint256 tdotAmount = IERC20(tdot).balanceOf(address(this));
64         uint256 wtdotAmount = this.totalSupply();
65
66         if (wtdotAmount == 0) {
67             return 1e18;
68         } else if (tdotAmount == 0) {
69             return 0;
70         } else {
71             return wtdotAmount.mul(1e18).div(tdotAmount);
72         }
73     }

```

Listing 3.1: `WrappedTDOT::depositRate()`

Recommendation Revise the above logic to simply the `depositRate` calculation.

Status The issue has been addressed by the following commit: [32d6ee3](#).

3.2 Possible Costly WTDOT From Improper Initialization

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `WrappedTDOT`
- Category: Time and State [\[6\]](#)
- CWE subcategory: CWE-362 [\[3\]](#)

Description

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

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

```

88     function deposit(uint256 tdotAmount) public returns (uint256) {
89         uint256 wtdotAmount = tdotAmount.mul(depositRate()).div(1e18);
90         require(wtdotAmount != 0, "WTDOT: invalid WTDOT amount");

92         IERC20(tdot).safeTransferFrom(msg.sender, address(this), tdotAmount);
93         _mint(msg.sender, wtdotAmount);

95         emit Deposit(msg.sender, tdotAmount, wtdotAmount);
96         return wtdotAmount;
97     }

```

Listing 3.2: `WrappedTDOT::deposit()`

```

62     function depositRate() public view returns (uint256) {
63         uint256 tdotAmount = IERC20(tdot).balanceOf(address(this));
64         uint256 wtdotAmount = this.totalSupply();
65
66         if (wtdotAmount == 0) {
67             return 1e18;
68         } else if (tdotAmount == 0) {
69             return 0;
70         } else {
71             return wtdotAmount.mul(1e18).div(tdotAmount);
72         }

```

Listing 3.3: `WrappedTDOT::depositRate()`

Specifically, when the pool is being initialized (line 66), the share value directly takes the value of `tdotAmount` (line 89), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `wdotAmount = tdotAmount = 1 WEI`. With that, the actor can further deposit a huge amount of the underlying assets with the goal of making the pool share extremely expensive.

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

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

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

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

3.3 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [7]
- CWE subcategory: CWE-561 [4]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested

manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the DAO [12] exploit, and the Uniswap/Lendf.Me hack [11].

We notice there are occasions where the checks-effects-interactions principle is violated. Using the `Staking` as an example, the `stake()` function (see the code snippet below) is provided to externally call a token contract to transfer provided tokens. However, the invocation of an external contract requires extra care in avoiding the above re-entrancy. For example, the interaction with the external contract (line 298) start before effecting the update on internal states, hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching re-entrancy via the same entry function.

```

287     function stake(uint256 poolId, uint256 amount)
288     public
289     virtual
290     override
291     updateRewards(poolId, msg.sender)
292     returns (bool)
293     {
294         require(amount > 0, "cannot stake 0");
295         IERC20 shareType = shareTypes(poolId);
296         require(address(shareType) != address(0), "invalid pool");
297
298         shareType.safeTransferFrom(msg.sender, address(this), amount);
299
300         _totalShares[poolId] = _totalShares[poolId].add(amount);
301         _shares[poolId][msg.sender] = _shares[poolId][msg.sender].add(amount);
302
303         emit Stake(msg.sender, poolId, amount);
304
305         return true;
306     }

```

Listing 3.4: `Staking::stake()`

While the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for re-entrancy, it is important to take precautions to thwart possible re-entrancy. Also, the current functions have the associated `nonReentrant` modifier, which eliminate the issue. However, we still feel the need of following the best practice of checks-effects-interactions.

Recommendation Apply necessary reentrancy prevention by following the checks-effects-interactions principle to block possible re-entrancy. Similarly, other contracts, e.g., `WrappedTDOT`, `UpgradeableStakingCommon`, and `UpgradeableStakingLST`, can be improved to make use of the `nonReentrant` modifier in their public functions, including `stake()`, `unstake()`, `deposit()`, and `withdraw()`.

Status The issue has been addressed by the following commit: 32d6ee3.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the Euphrates contracts, 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.

```

27     function pause() external onlyOwner {
28         _pause();
29     }

31     /// @notice Unpause the contract by Pausable.
32     /// @dev Define the 'onlyOwner' access.
33     function unpause() external onlyOwner {
34         _unpause();
35     }

37     /// @inheritdoc PoolOperationPausable
38     /// @dev Override the inherited function to define 'onlyOwner' and 'whenNotPaused'
39     ///      access.
40     function setPoolOperationPause(uint256 poolId, Operation operation, bool paused)
41     public
42     override
43     onlyOwner
44     whenNotPaused
45     {
46         super.setPoolOperationPause(poolId, operation, paused);
47     }

48     /// @inheritdoc Staking
49     /// @dev Override the inherited function to define 'onlyOwner' and 'whenNotPaused'
50     ///      access.
51     function addPool(IERC20 shareType) public override onlyOwner whenNotPaused {
52         super.addPool(shareType);

```

Listing 3.5: 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 mitigated 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 protocol. This support is for DOT LSD farming that will deploy at Acala EVM+. The contract can convert assets (LcDOT /DOT) of the pool into LSD assets (LDOT/tDOT) by external call the system contracts (LiquidCrowdloan, Homa, and StableAssets) which bridge the pallet module of the Acala runtime. 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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). <https://cwe.mitre.org/data/definitions/362.html>.
- [4] MITRE. CWE-561: Dead Code. <https://cwe.mitre.org/data/definitions/561.html>.
- [5] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [6] MITRE. CWE CATEGORY: 7PK - Time and State. <https://cwe.mitre.org/data/definitions/361.html>.
- [7] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.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>.

- [11] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.
- [12] David Siegel. Understanding The DAO Attack. <https://www.coindesk.com/understanding-dao-hack-journalists>.

