SMART CONTRACT AUDIT REPORT

for

LendFlare AutoCompounding

Prepared By: Xiaomi Huang

PeckShield

May 23, 2022

Document Properties

Client	Lend Flare
Title	Smart Contract Audit Report
Target	LendFlare AutoCompounding
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 23, 2022	Xuxian Jiang	Final Release
1.0-rc	May 23, 2022	Xuxian Jiang	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 design document and related smart contract source code of the `AutoCompounding` support of the `LendFlare` 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 audited protocol 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 Lend Flare

The `Lend Flare` protocol is a decentralized borrowing platform on `Ethereum` that allows `Curve LP` holders (the borrowers) to draw fixed-rate, fixed term and high LTV loans against `Curve LP` tokens used as collaterals, with no concerns for assets being liquidated due to price fluctuation. Loans are paid out through `Lend Flare` from the `Compound` platform. Liquidity providers (the lenders) who deposit loan liquidity through `Lend Flare` on `Compound` will receive one of the highest interest rate compared to other current lending platforms. The audited `AutoCompounding` protocol provides extra rewards for the staking users. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of LendFlare AutoCompounding

Item	Description
Name	Lend Flare
Website	https://lendflare.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 23, 2022

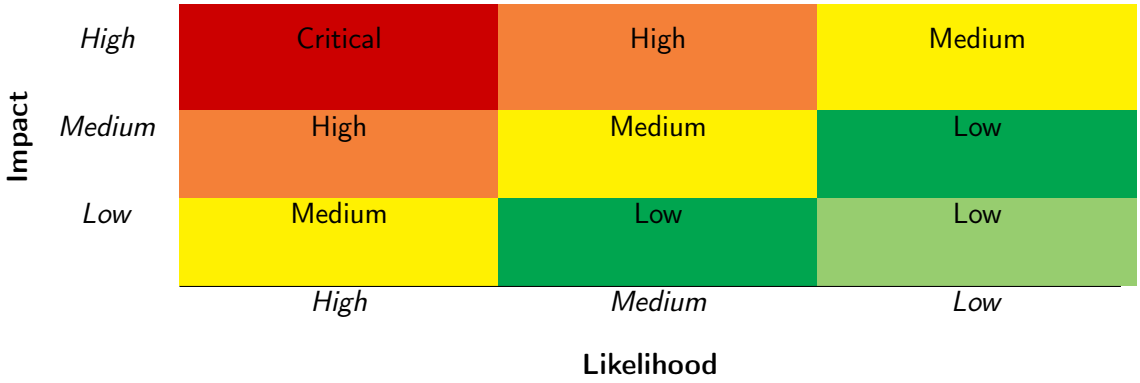
In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/LendFlare/lendflare-auto-compounding.git (f2e290e)

1.2 About PeckShield

PeckShield Inc. [7] 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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [6]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

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

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [5], 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 `LendFlare AutoCompounding` 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 logic, 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	1	■
Informational	0	
Total	2	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability and 1 low-severity vulnerability.

Table 2.1: Key LendFlare AutoCompounding Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Costly lfCRV From Improper Initialization	Time And State	Confirmed
PVE-002	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 Possible Costly lfCRV From Improper Initialization

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LendFlareCRV`
- Category: Time and State [4]
- CWE subcategory: CWE-362 [2]

Description

The `AutoCompounding` support 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.

```
181   function _deposit(address _recipient, uint256 _amount) internal returns (uint256) {
182     require(_amount > 0, "LendFlareCRV: zero amount deposit");
183     uint256 _underlying = totalUnderlying();
184     uint256 _totalSupply = totalSupply();

186     IERC20Upgradeable(CVXCRV).safeApprove(CVXCRV_STAKING, 0);
187     IERC20Upgradeable(CVXCRV).safeApprove(CVXCRV_STAKING, _amount);
188     IConvexBasicRewards(CVXCRV_STAKING).stake(_amount);

190     uint256 _shares;
191     if (_totalSupply == 0) {
192       _shares = _amount;
193     } else {
194       _shares = _amount.mul(_totalSupply) / _underlying;
195     }
196     _mint(_recipient, _shares);
```

```
198     emit Deposit(msg.sender, _recipient, _amount);
199     return _shares;
200   }
```

Listing 3.1: `LendFlareCRV::_deposit()`

Specifically, when the pool is being initialized (line 191), the share value directly takes the value of `_amount` (line 192), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `_shares = _amount = 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 confirmed and the team plans to follow a guarded launch so that a trusted user will be the first to deposit.

3.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `AutoCompounding` support of the `Lend Flare` protocol, there is a special administrative account `owner`. This account plays a critical role in governing and regulating the protocol-wide operations

(e.g., configure parameters and execute privileged operations). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that this privileged account needs to be scrutinized. In the following, we examine its related privileged accesses in current protocol.

```
537     function updateSwap(address _zap) external onlyOwner {
538         require(_zap != address(0), "LendingBooster: zero zap address");
539         zap = _zap;
540
541         emit UpdateZap(_zap);
542     }
543
544     function pausePoolWithdraw(uint256 _pid, bool _status) external onlyOwner {
545         require(_pid < poolInfo.length, "LendingBooster: invalid pool");
546
547         poolInfo[_pid].pauseWithdraw = _status;
548
549         emit PausePoolWithdraw(_pid, _status);
550     }
551
552     function pausePoolDeposit(uint256 _pid, bool _status) external onlyOwner {
553         require(_pid < poolInfo.length, "LendingBooster: invalid pool");
554
555         poolInfo[_pid].pauseDeposit = _status;
556
557         emit PausePoolDeposit(_pid, _status);
558     }
```

Listing 3.2: Example Privileged Operations in `LendFlareVault`

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 Promptly transfer the administrative privileges to the intended `DAO`-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been confirmed by the team. The team clarify they will use a multi-sig account as the `owner`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the AutoCompounding feature of the Lend Flare protocol, which is a decentralized borrowing platform on Ethereum that allows Curve LP holders (the borrowers) to draw fixed-rate, fixed term and high LTV loans against Curve LP tokens used as collaterals, with no concerns for assets being liquidated due to price fluctuation. The AutoCompounding feature provides extra rewards for the staking users. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

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

[4] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

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