SMART CONTRACT AUDIT REPORT

for

Lend Flare

Prepared By: Xiaomi Huang

PeckShield

May 15, 2022

Document Properties

Client	Lend Flare
Title	Smart Contract Audit Report
Target	Lend Flare
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 15, 2022	Xuxian Jiang	Final Release
1.0-rc	May 12, 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 Lend Flare 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 basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Lend Flare

Item	Description
Name	Lend Flare
Website	https://lendflare.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 15, 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_finance_contracts_v2.git (08fb645)

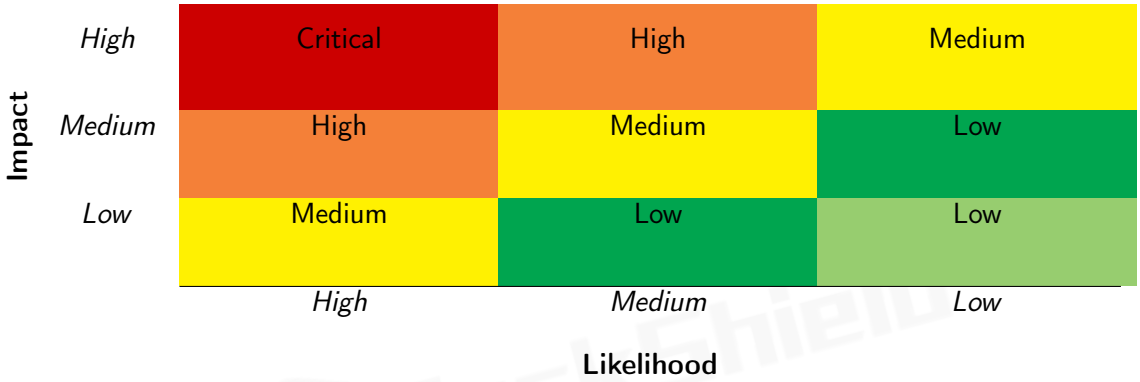
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/LendFlare/lendflare_finance_contracts_v2.git (fe20a31)

1.2 About PeckShield

PeckShield Inc. [12] 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 [11]:

- 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) [10], 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 `Lend Flare` 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	2	■■
Low	3	■■■
Informational	2	■■
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 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 2 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key Lend Flare Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Sandwich/MEV For Reduced Return	Time And State	Confirmed
PVE-002	Informational	Simplified Logic in getReward()	Business Logic	Resolved
PVE-003	Informational	Redundant State/Code Removal	Coding Practices	Resolved
PVE-004	Low	Revisited Initialization in SupplyTreasuryFundForCompoundV2	Business Logic	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-006	Medium	Proper Liquidation Order in ConvexBoosterV2::liquidate()	Business Logic	Resolved
PVE-007	Low	Proper Liquidate Amount Enforcement in LendingMarketV2	Business Logic	Resolved

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 Sandwich/MEV For Reduced Return

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `ConvexBoosterV2`
- Category: Time and State [9]
- CWE subcategory: CWE-663 [4]

Description

The `Lend Flare` protocol is a decentralized borrowing platform. Naturally, if the borrower's loan is past due, the protocol will liquidate assets by swapping the `Curve LP` that has been collateralized to the target asset. While analyzing the liquidation logic, we notice the current implementation can be improved.

Specifically, we show below the related `_removeLiquidity()` function. This function implements a rather straightforward logic in removing the liquidity with the collateralized `Curve LP` (line 488). However, it comes to our attention that the removed liquidity is converted to the target asset with the call `remove_liquidity_one_coin()`, which does not have any slippage control. With that, it is possible to have a MEV issue to imbalance the pool and take advantage of the liquidation for profit!

```
479     function _removeLiquidity(
480         address _lpToken,
481         address _curveSwapAddress,
482         uint256 _amount,
483         int128 _coinId
484     ) internal {
485         if (metaPoolInfo[_lpToken].zapAddress != address(0)) {
486             if (metaPoolInfo[_lpToken].isMetaFactory) {
487                 ICurveSwapV2(metaPoolInfo[_lpToken].zapAddress)
488                     .remove_liquidity_one_coin(_lpToken, _amount, _coinId, 0);
489
490                 emit RemoveLiquidity(
491                     _lpToken,
```

```
492                     _curveSwapAddress,
493                     _amount,
494                     _coinId
495                 );
496
497                 return;
498             }
499         }
500
501         ICurveSwapV2(_curveSwapAddress).remove_liquidity_one_coin(
502             _amount,
503             _coinId,
504             0
505         );
506
507         emit RemoveLiquidity(_lpToken, _curveSwapAddress, _amount, _coinId);
508     }
```

Listing 3.1: `ConvexBoosterV2::_removeLiquidity()`

Recommendation Add necessary slippage control to avoid being sandwiched to fairly liquidate the collateralized asset.

Status This issue has been confirmed.

3.2 Simplified Logic in getReward()

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `BaseReward, ConvexRewardPool`
- Category: Business Logic [8]
- CWE subcategory: CWE-770 [5]

Description

The `Lend Flare` protocol has the built-in incentive mechanism with the `BaseReward` and `ConvexRewardPool` contracts. While examining these two contracts, we notice that the `getReward()` routine is intended to obtain the calling user's staking rewards. The logic is rather straightforward in calculating possible reward, which, if not zero, is then allocated to the calling (staking) user.

Our examination shows that the current implementation logic can be further optimized. In particular, the `getReward()` routine has a modifier, i.e., `updateReward()`, which timely updates the queried user's (earned) rewards in `rewards[msg.sender]` (line 755).

```
147     function getReward(address _for)
148         public
```

```
149         override
150         nonReentrant
151         updateReward(_for)
152     {
153         uint256 reward = earned(_for);

155         if (reward > 0) {
156             rewards[_for] = 0;

158             if (rewardToken != address(0)) {
159                 IERC20(rewardToken).safeTransfer(_for, reward);
160             } else {
161                 require(
162                     address(this).balance >= reward,
163                     "BaseReward: !address(this).balance"
164                 );

166                 payable(_for).sendValue(reward);
167             }

169             emit RewardPaid(_for, reward);
170         }
171     }
```

Listing 3.2: `BaseReward::getReward()`

```
48     modifier updateReward(address account) {
49         rewardPerTokenStored = rewardPerToken();
50         lastUpdateTime = lastTimeRewardApplicable();

52         if (account != address(0)) {
53             rewards[account] = earned(account);
54             userRewardPerTokenPaid[account] = rewardPerTokenStored;
55         }
56         _;
57     }
```

Listing 3.3: `BaseReward::updateReward()`

Having the modifier `updateReward()`, there is no need to re-calculate the earned reward for the given account `_for`. In other words, we can simply re-use the calculated `rewards[_for]` and assign it to the `reward` variable (line 153).

Recommendation Avoid the duplicated calculation of the caller's reward in `getReward()`, which also leads to (small) beneficial reduction of associated gas cost.

```
147     function getReward(address _for)
148         public
149         override
150         nonReentrant
151         updateReward(_for)
152     {
```

```
153         uint256 reward = rewards[_for];

155         if (reward > 0) {
156             rewards[_for] = 0;

158             if (rewardToken != address(0)) {
159                 IERC20(rewardToken).safeTransfer(_for, reward);
160             } else {
161                 require(
162                     address(this).balance >= reward,
163                     "BaseReward: !address(this).balance"
164                 );

166                 payable(_for).sendValue(reward);
167             }

169             emit RewardPaid(_for, reward);
170         }
171     }
```

Listing 3.4: Revised `BaseReward::getReward()`

Status This issue has been fixed in the commit: fe20a31.

3.3 Redundant State/Code Removal

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `SupplyTreasuryFund`
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [3]

Description

The `Lend Flare` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, and `ReentrancyGuard`, to facilitate its code implementation and organization. For example, the `SupplyTreasuryFund` smart contract has so far imported at least three reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `SupplyTreasuryFund` contract, there is a setter function `setBaseRate()` to configure the base rate. However, it comes to our attention that there are two validation checks. The first one validates the given value is smaller than a constant value 23782343988 (line 271) and the second validation ensures the given value is non-negative (line 272). Note that the second validation is basically a no-op and can be safely removed.

```
270     function setBaseRate(uint256 _v) external onlyGovernance {
271         require(_v < 23782343988, "!_v");
272         require(_v >= 0, "!_v");
273
274         baseRate = _v;
275     }
```

Listing 3.5: SupplyTreasuryFund::setBaseRate()

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status This issue has been fixed in the commit: fe20a31.

3.4 Revisited Initialization in SupplyTreasuryFundForCompoundV2

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: SupplyTreasuryFundForCompoundV2
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

The Lend Flare protocol has a SupplyTreasuryFundForCompoundV2 contract that takes the liquidity provided by lenders to stake in Compound. In case if no borrow occurs, lenders could still earn full interest from Compound. When there is a borrow, lenders will be able to have a base supply interest higher than Compound. When analyzing the contract's initialization logic, we notice the current implementation can be improved.

To elaborate, we show below the related function initialize(). It properly initializes various state variables, including underlyToken, virtualBalance, isErc20, rewardCompPool and frozenUnderlyToken. However, the compAddress state may not be initialized when the oldSupplyTreasury is currently not configured, i.e., oldSupplyTreasury == address(0). Therefore, in either case, there is a need to properly initialize the compAddress state. Otherwise, the claim() function may not work as expected.

```
173     function initialize(
174         address _virtualBalance,
175         address _underlyToken,
176         bool _isErc20
177     ) public onlyOwner {
178         require(!initialized, "initialized");
```

```
180         underlyingToken = _underlyingToken;

182         virtualBalance = _virtualBalance;
183         isErc20 = _isErc20;

185         if (oldSupplyTreasury == address(0)) {
186             compAddress = ICompoundComptroller(compoundComptroller)
187                 .getCompAddress();

189             rewardCompPool = ISupplyRewardFactory(supplyRewardFactory)
190                 .createReward(compAddress, virtualBalance, address(this));
191         } else {
192             rewardCompPool = IOldSupplyTreasury(oldSupplyTreasury)
193                 .rewardCompPool();
194             frozenUnderlyToken = IOldSupplyTreasury(oldSupplyTreasury)
195                 .frozenUnderlyToken();
196         }

198         initialized = true;
199     }
```

Listing 3.6: `SupplyTreasuryFundForCompoundV2::initialize()`

Recommendation Properly initialize the `compAddress` state in the above `SupplyTreasuryFundForCompoundV2` contract.

Status This issue has been fixed in the commit: fe20a31.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In 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.

```
224     function migrate(address _newTreasuryFund, bool _setReward)
```

```
225         external
226         onlyOwner
227         nonReentrant
228         returns (uint256)
229     {
230         uint256 cTokens = IERC20(lpToken).balanceOf(address(this));

232         uint256 redeemState = ICompound(lpToken).redeem(cTokens);

234         require(
235             redeemState == 0,
236             "SupplyTreasuryFundForCompound: !redeemState"
237         );

239         uint256 bal;

241         if (isErc20) {
242             bal = IERC20(underlyToken).balanceOf(address(this));

244             sendToken(underlyToken, owner, bal);
245         } else {
246             bal = address(this).balance;

248             if (bal > 0) {
249                 sendToken(address(0), owner, bal);
250             }
251         }

253         if (_setReward) {
254             IBaseReward(rewardCompPool).addOwner(_newTreasuryFund);
255             IBaseReward(rewardCompPool).removeOwner(address(this));
256         }

258         return bal;
259     }
```

Listing 3.7: `SupplyTreasuryFundForCompoundV2::migrate()`

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`.

3.6 Proper Liquidation Order in ConvexBoosterV2::liquidate()

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `ConvexBoosterV2`
- Category: Business Logic [8]
- CWE subcategory: CWE-770 [5]

Description

As mentioned earlier, as a decentralized borrowing platform, the `Lend Flare` protocol will liquidate if a borrower's loan is past due. While analyzing the liquidation logic, we notice there is a need to properly claim the rewards for the liquidated borrower and the current implementation can be improved.

To elaborate, we show below the related `liquidate()` function from the `ConvexBoosterV2` contract. It properly claims various rewards from `rewardCrvPool` and `rewardCvxPool`, then removes the collaterized `Curve LPs`, and sends the assets back to `LendingMarketV2`. It comes to our attention that the rewards are claimed after deducting the balance from the `pool.virtualBalance` (line 529), which may reduce the rewards entitled to the borrower being liquidated. To correct, the balance reduction should be performed after claiming the rewards, instead of before!

```
510     function liquidate(
511         uint256 _pid,
512         int128 _coinId,
513         address _user,
514         uint256 _amount
515     )
516         external
517         override
518         onlyLendingMarket
519         nonReentrant
520         returns (address, uint256)
521     {
522         PoolInfo storage pool = poolInfo[_pid];

524         IOriginConvexRewardPool(pool.originCrvRewards).withdrawAndUnwrap(
525             _amount,
526             true
527         );

529         IVirtualBalanceWrapper(pool.virtualBalance).withdrawFor(_user, _amount);

531         if (IConvexRewardPool(pool.rewardCrvPool).earned(_user) > 0) {
532             IConvexRewardPool(pool.rewardCrvPool).getReward(_user);
533         }
```

```
535         if (IConvexRewardPool(pool.rewardCvxPool).earned(_user) > 0) {
536             IConvexRewardPool(pool.rewardCvxPool).getReward(_user);
537         }

539         IConvexRewardPool(pool.rewardCrvPool).withdraw(_user);
540         IConvexRewardPool(pool.rewardCvxPool).withdraw(_user);

542         IERC20(pool.lpToken).safeApprove(pool.curveSwapAddress, 0);
543         IERC20(pool.lpToken).safeApprove(pool.curveSwapAddress, _amount);

545         address underlyToken;

547         if (metaPoolInfo[pool.lpToken].zapAddress != address(0)) {
548             if (
549                 metaPoolInfo[pool.lpToken].swapAddress ==
550                 metaPoolInfo[pool.lpToken].basePoolAddress
551                 (!metaPoolInfo[pool.lpToken].isMeta &&
552                     !metaPoolInfo[pool.lpToken].isMetaFactory)
553                 _coinId == 0
554             ) {
555                 underlyToken = _coins(pool.curveSwapAddress, _coinId);
556             } else {
557                 underlyToken = _coins(
558                     metaPoolInfo[pool.lpToken].basePoolAddress,
559                     _coinId - 1
560                 );
561             }
562         } else {
563             underlyToken = _coins(pool.curveSwapAddress, _coinId);
564         }

566         _removeLiquidity(pool.lpToken, pool.curveSwapAddress, _amount, _coinId);

568         if (underlyToken == ZERO_ADDRESS) {
569             uint256 totalAmount = address(this).balance;

571             msg.sender.sendValue(totalAmount);

573             return (ZERO_ADDRESS, totalAmount);
574         } else {
575             uint256 totalAmount = IERC20(underlyToken).balanceOf(address(this));

577             IERC20(underlyToken).safeTransfer(msg.sender, totalAmount);

579             return (underlyToken, totalAmount);
580         }
581     }
```

Listing 3.8: `ConvexBoosterV2::liquidate()`

Recommendation Properly claim the rewards before deducting the balance from the liquidated borrower.

Status This issue has been fixed in the commit: fe20a31.

3.7 Proper Liquidate Amount Enforcement in LendingMarketV2

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LendingMarketV2`
- Category: Business Logic [8]
- CWE subcategory: CWE-770 [5]

Description

In last section, we have examined the reward-claiming logic when a liquidation occurs. In this section, we further analyze the liquidation logic and examine the calculation of the liquidation amount.

To elaborate, we show below the related `liquidate()` function from the `LendingMarketV2` contract. We notice that if the valued LP is not sufficient to cover the debt, the current implementation supports the use of `msg.value` or `_extraErc20Amount` to force the liquidation. However, we notice the current logic does not validate the `msg.value` or `_extraErc20Amount`. As a result, if the given amount of `msg.value` or `_extraErc20Amount` plus the `liquidateAmount` may not be sufficient to cover the debt, the current logic still allows the liquidation to proceed.

```
580     function liquidate(bytes32 _lendingId, uint256 _extraErc20Amount)
581         public
582         payable
583         nonReentrant
584     {
585         uint256 gasStart = gasleft();
586         LendingInfo storage lendingInfo = lendings[_lendingId];

588         require(lendingInfo.startedBlock > 0, "!invalid lendingId");

590         UserLending storage userLending = userLendings[lendingInfo.user][
591             lendingInfo.userLendingIndex
592         ];

594         require(
595             lendingInfo.state == UserLendingState.LENDING,
596             "!UserLendingState"
597         );

599         require(
600             lendingInfo.startedBlock.add(userLending.borrowNumbers).sub(
601                 liquidateThresholdBlockNumbers
602             ) < block.number,
603             "!borrowNumbers"
604         );
```

```
606         PoolInfo storage pool = poolInfo[lendingInfo.pid];

608         lendingInfo.state = UserLendingState.LIQUIDATED;

610         BorrowInfo storage borrowInfo = borrowInfos[
611             generateId(address(0), lendingInfo.pid, userLending.supportPid)
612         ];

614         borrowInfo.borrowAmount = borrowInfo.borrowAmount.sub(
615             userLending.token0Price
616         );
617         borrowInfo.supplyAmount = borrowInfo.supplyAmount.sub(
618             userLending.lendingAmount
619         );

621         Statistic storage statistic = myStatistics[
622             generateId(
623                 lendingInfo.user,
624                 lendingInfo.pid,
625                 userLending.supportPid
626             )
627         ];

629         statistic.totalCollateral = statistic.totalCollateral.sub(
630             userLending.token0
631         );
632         statistic.totalBorrow = statistic.totalBorrow.sub(
633             userLending.lendingAmount
634         );

636         (address underlyToken, uint256 liquidateAmount) = IConvexBooster(
637             convexBooster
638         ).liquidate(
639                 pool.convexPid,
640                 userLending.curveCoinId,
641                 lendingInfo.user,
642                 userLending.token0
643             );

645         if (underlyToken == ZERO_ADDRESS) {
646             liquidateAmount = liquidateAmount.add(msg.value);

648             ISupplyBooster(supplyBooster).liquidate{value: liquidateAmount}(
649                 userLending.lendingId,
650                 userLending.borrowInterest
651             );
652         } else {
653             IERC20(underlyToken).safeTransfer(supplyBooster, liquidateAmount);

655             if (_extraErc20Amount > 0) {
656                 // Failure without authorization
```

```
657                 IERC20(underlyToken).safeTransferFrom(
658                     msg.sender,
659                     supplyBooster,
660                     _extraErc20Amount
661                 );
662             }
663
664             ISupplyBooster(supplyBooster).liquidate(
665                 userLending.lendingId,
666                 userLending.borrowInterest
667             );
668         }
669
670         ILendingSponsor(lendingSponsor).payFee(
671             userLending.lendingId,
672             msg.sender
673         );
674
675         uint256 gasSpent = (21000 + gasStart - gasleft()).mul(tx.gasprice);
676
677         emit Liquidate(
678             userLending.lendingId,
679             lendingInfo.user,
680             liquidateAmount,
681             gasSpent,
682             lendingInfo.state
683         );
684     }
```

Listing 3.9: `LendingMarketV2::liquidate()`

Recommendation Properly validate the sum of `liquidateAmount` and the given `msg.value` or `_extraErc20Amount` so that the debt is fully covered.

Status This issue has been fixed in the commit: fe20a31.

4 | Conclusion

In this audit, we have analyzed the design and implementation 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. Loans are paid out through Lend Flare from the Compound platform. Liquidity providers (the lenders) who deposit loan liquidity through Lendflare on Compound will receive one of the highest interest rate compared to other current lending platforms. 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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[5] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.html.

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

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

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

[9] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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