SMART CONTRACT AUDIT REPORT

for

ExtraFi

Prepared By: Xiaomi Huang

PeckShield

May 5, 2023

Document Properties

Client	ExtraFi
Title	Smart Contract Audit Report
Target	ExtraFi
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Patrick Lou, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 5, 2023	Xuxian Jiang	Final Release
1.0-rc	April 30, 2023	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 Extra Finance (ExtraFi) 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 ExtraFi

Extra Finance is a community-driven leveraged yield farming (LYF) protocol built on Optimism. By offering up to 3X leverage, Extra Finance enables users to farm a diverse range of farming pools on Velodrome and other DEXes. Users can customize their farming strategies with options like re-investing, market-neutral, and long/short farming strategies. In addition to LYF, Extra Finance also functions as a lending protocol. Users can deposit funds into its lending pools to earn interest on their deposited assets. This feature provides users with a way to earn passive income. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of ExtraFi

Item	Description
Target	ExtraFi
Website	https://extrafi.io/
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 5, 2023

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

- https://github.com/ExtraFi/contracts.git (c70f83d)

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

- https://github.com/ExtraFi/contracts.git (ebed8b1)

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

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 [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.

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 1.3: 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

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 ExtraFi 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	1	■
Medium	1	■
Low	3	■ ■ ■
Informational	1	■
Total	6	

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 high-severity vulnerability, 1 medium-severity vulnerability, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key ExtraFi Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Possible Position Value Manipulation	Time And State	Resolved
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-003	Low	Incorrect Balance Calculation in VeToken::balanceOfAt()	Coding Practices	Resolved
PVE-004	Low	Improved Logic in StakingRewards::setReward()	Business Logic	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-006	Informational	Redundant State/Code Removal	Coding Practices	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 Position Value Manipulation

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: Pool
- Category: Time and State [8]
- CWE subcategory: CWE-682 [4]

Description

As mentioned earlier, ExtraFi is a leveraged yield farming (LYF) protocol that enables users to farm a diverse range of farming pools on Velodrome and other DEXes. Therefore, the protocol has a common need to ensure the farming position remains healthy. For that, there is a validatePositionLeverage() helper. While examining this helper, we notice the current approach to evaluate a position value might be manipulated.

In the following, we show the VeloPositionValue::validatePositionLeverage() routine. As the name indicates, this routine is designed to examine the liquidity (with the associated value) owned by the position. It also computes the respective debt value and evaluate the resulting leverage will not exceed the maximum allowed leverage. However, the liquidity of the given position is directly used to calculate the token amount and the value (by multiplying the TWAP price). And the liquidity-derived token amount is directly computed from the instant reserve, which unfortunately may suffer from flashloan for manipulation.

```
    function validatePositionLeverage(uint256 positionId) internal view {
        VaultTypes.VeloVaultStorage storage vaultStorage = StateAccessor
            .getVaultStorage();
        VaultTypes.VeloVaultState storage vaultState = vaultStorage.state;
        VaultTypes.VeloPosition storage position = StateAccessor.getPosition(
            vaultStorage,
            positionId
        );

```

```
        validateLeverageState memory state;
        state.liquidity = position.lpShares.mul(vaultState.totalLp).div(
            vaultState.totalLpShares
        );

        (state.amount0, state.amount1) = getLiquidityUnderlingTokens(
            state.liquidity
        );
        (state.amount0, state.amount1) = (
            state.amount0.add(position.token0Left),
            state.amount1.add(position.token1Left)
        );

        state.price = getTwapPrice();
        state.totalValue = valueOfTokensInToken0(
            state.amount0,
            state.amount1,
            state.price
        );

        (state.debt0, state.debt1) = DebtLogic.debtOfVaultPosition(
            vaultState,
            position,
            ILendingPool(vaultStorage.lendingPool)
        );
        state.debtValue = valueOfTokensInToken0(
            state.debt0,
            state.debt1,
            state.price
        );

        uint16 leverage = VeloVaultPremium.isPremium(position.manager)
            ? vaultState.premiumMaxLeverage
            : vaultState.maxLeverage;

        require(
            state.totalValue > state.debtValue &&
                state.totalValue.sub(state.debtValue).mul(leverage) >=
                state.totalValue.mul(100),
            "OOL"
        );
    }
```

Listing 3.1: `VeloPositionValue::validatePositionLeverage()`

Specifically, a malicious actor may intentionally perform a large swap with flashloan funds to make the target pair pool imbalanced and then borrow with the maximum allowed leverage, which will succeed since the imbalanced pool leads to the inflation of computed liquidity value. After that, the actor performs a reverse swap to profit. Note the reserve swap will immediately put the borrow position underwater once the (inflated) liquidity value is deflated, hence resulting in the protocol loss.

This issue is in essence related to the LP token pricing and can be better mitigate with the fair reserve approach as elaborated in https://blog.alphaventuredao.io/fair-lp-token-pricing/.

Recommendation Develop a robust LP token pricing approach to evaluate the liquidity value.

Status The issue has been fixed by this commit: ebed8b1.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Business Logic [7]
- CWE subcategory: N/A

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the transfer() routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., USDT, as our example. We show the related code snippet below. Specifically, the transfer() routine does not have a return value defined and implemented. However, the IERC20 interface has defined the transfer() interface with a bool return value. As a result, the call to transfer() may expect a return value. With the lack of return value of USDT's transfer(), the call will be unfortunately reverted.

```
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 3.2: USDT::transfer()

Because of that, a normal call to transfer() is suggested to use the safe version, i.e., safeTransfer (), In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful.

In current implementation, if we examine the `StakingRewards::withdrawByLendingPool()` routine that is designed to withdraw certain amount of staked assets from the `StakingRewards` contract. To accommodate the specific idiosyncrasy, there is a need to user `safeTransfer()`, instead of `transfer()` (line 149).

```
    function withdrawByLendingPool(
        uint amount,
        address user,
        address to
    ) external onlyLendingPool nonReentrant updateReward(user) {
        require(amount > 0, "amount = 0");

        balanceOf[user] -= amount;
        totalStaked -= amount;

        require(stakedToken.transfer(to, amount), "transfer failed");

        emit Withdraw(user, to, amount);
    }
```

Listing 3.3: `StakingRewards::withdrawByLendingPool()`

Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`. This issue is present in a number of contracts and their functions.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`. Note the `safeApprove()` needs to be performed twice: the first resets the spending allowance and the second approves the intended amount.

Status The issue has been fixed by this commit: `ebed8b1`.

3.3 Incorrect Balance Calculation in VeToken::balanceOfAt()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `VeToken`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

The `ExtraFi` protocol has a governance-oriented `VeToken` contract, which keeps track of the voting power of each participant. While examining an internal helper, we notice its current implementation needs to be improved.

To elaborate, we show below the related `VeToken::balanceOfAt()` helper. It has a dedicated purpose in calculating the voting power of the given user at the specified `blockNumber`. Note the voting power is extrapolated from the neighboring checkpoints. And the extrapolation requires the accurate `dBlock` and `dt` (lines 380-387). And the `dBlock` (line 387) is currently computed as `blockNumber - point0.blk`, which needs to be revised as `block.number - point0.blk`.

```
350     function balanceOfAt(
351         address addr,
352         uint256 blockNumber
353     ) public view returns (uint256) {
354         uint256 min = 0;
355         uint256 max = userPointEpoch[addr];
356 
357         // Find the approximate timestamp for the block number
358         for (uint256 i = 0; i < 128; i++) {
359             if (min >= max) {
360                 break;
361             }
362             uint256 mid = (min + max + 1) / 2;
363             if (userPointHistory[addr][mid].blk <= blockNumber) {
364                 min = mid;
365             } else {
366                 max = mid - 1;
367             }
368         }
369 
370         // min is the userEpoch nearest to the block number
371         Point memory uPoint = userPointHistory[addr][min];
372         uint256 maxEpoch = epoch;
373 
374         // blocktime using the global point history
375         uint256 _epoch = _findBlockEpoch(blockNumber, maxEpoch);
376         Point memory point0 = pointHistory[_epoch];
377         uint256 dBlock = 0;
378         uint256 dt = 0;
379 
380         if (_epoch < maxEpoch) {
381             Point memory point1 = pointHistory[_epoch + 1];
382             dBlock = point1.blk - point0.blk;
383             dt = point1.ts - point0.ts;
384         } else {
385             dBlock = blockNumber - point0.blk;
386             dt = block.timestamp - point0.ts;
387         }
388 
389         uint256 blockTime = point0.ts;
390         if (dBlock != 0) {
391             blockTime += (dt * (blockNumber - point0.blk)) / dBlock;
392         }
393 
```

```
        uPoint.bias -=
            uPoint.slope *
            int128(int256(blockTime) - int256(uPoint.ts));
        if (uPoint.bias < 0) {
            uPoint.bias = 0;
        }
        return uint256(int256(uPoint.bias));
    }
```

Listing 3.4: `VeToken::balanceOfAt()`

Recommendation Correct the above implementation to properly compute the user voting power.

Status The issue has been fixed by this commit: `ebed8b1`.

3.4 Improved Logic in StakingRewards::setReward()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `StakingRewards`
- Category: Numeric Errors [9]
- CWE subcategory: CWE-190 [1]

Description

The `ExtraFi` protocol has a `StakingRewards` contract to incentivize protocol users. In the process of examining the logic to add new rewards, we notice the current implementation leaves a corner case unaddressed.

To elaborate, we show below the related function `setReward()`. This function basically adds additional rewards into the pool. However, there is a corner case, i.e., `block.timestamp > startTime and totalStaked==0)`. When it occurs, the given `startTime` is already passed and there is no stake. As a result, the reward amount accumulated in the time period of `[startTime, block.timestamp]` is not accounted for.

```
    function setReward(
        address rewardToken,
        uint256 startTime,
        uint256 endTime,
        uint256 totalRewards
    ) public onlyOwner nonReentrant updateReward(address(0)) {
        require(startTime < endTime, "start must lt end");
        require(rewardData[rewardToken].endTime < block.timestamp, "not end");

        if (!inRewardsTokenList[rewardToken]) {
```

```
            rewardTokens.push(rewardToken);
            inRewardsTokenList[rewardToken] = true;
        }

        rewardData[rewardToken].startTime = startTime;
        rewardData[rewardToken].endTime = endTime;
        rewardData[rewardToken].lastUpdateTime = block.timestamp;
        rewardData[rewardToken].rewardRate =
            totalRewards /
            (endTime - startTime);

        if (block.timestamp > startTime && totalStaked > 0) {
            uint256 dt = block.timestamp - startTime;

            rewardData[rewardToken].rewardPerTokenStored +=
                (rewardData[rewardToken].rewardRate * dt * 1e18) /
                totalStaked;
        }

        IERC20(rewardToken).transferFrom(
            msg.sender,
            address(this),
            totalRewards
        );

        emit RewardsSet(rewardToken, startTime, endTime, totalRewards);
    }
```

Listing 3.5: `StakingRewards::setReward()`

Recommendation Improve the above routine to ensure it addresses all possible corner cases.

Status The issue has been fixed by this commit: `ebed8b1`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `ExtraFi` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure various system parameters, create/manage new vaults, as well as set up rewards). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function adminSetVault(
        uint256 vaultId,
        bytes calldata params
    ) external nonReentrant onlyOwner {
        address vaultAddress = IVaultFactory(vaultFactory).vaults(vaultId);
        require(vaultAddress != address(0), Errors.VL_ADDRESS_CANNOT_ZERO);
        IVeloVault(vaultAddress).adminSetVault(params);
    }

    function enablePermissionLessLiquidation() public nonReentrant onlyOwner {
        permissionLessLiquidationEnabled = true;
    }

    function disablePermissionLessLiquidation() public nonReentrant onlyOwner {
        permissionLessLiquidationEnabled = false;
    }

    function addPermissionedLiquidator(
        address addr
    ) public nonReentrant onlyOwner {
        liquidatorWhitelist[addr] = true;
    }

    function removePermissionedLiquidator(
        address addr
    ) public nonReentrant onlyOwner {
        liquidatorWhitelist[addr] = false;
    }

    function enablePermissionLessCompound() public nonReentrant onlyOwner {
        permissionLessCompoundEnabled = true;
    }

    function disablePermissionLessCompound() public nonReentrant onlyOwner {
        permissionLessCompoundEnabled = false;
    }

    function addPermissionedCompounder(
        address addr
    ) public nonReentrant onlyOwner {
        compounderWhitelist[addr] = true;
    }

    function removePermissionedCompounder(
        address addr
    ) public nonReentrant onlyOwner {
        compounderWhitelist[addr] = false;
    }
```

Listing 3.6: Example Privileged Operations in `VeloPositionManager`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it would be worrisome if the privileged account is not governed by a DAO-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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 The issue has been confirmed by the team. The team intends to introduce multi-sig and `timelock` mechanisms to mitigate this issue.

3.6 Redundant State/Code Removal

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

While reviewing the implementation of `ExtraFi` protocol, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed. Using `LendingPool::redeem()` as an example, it is designed to redeem `eTokens` in exchange for the underlying asset. However, we observe this function has a number of modifiers and two or them can be simply removed, i.e., `payable` and `avoidUsingNativeEther`. The same issue is also applicable to another `unStakeAndWithdraw()` routine.

```
    function redeem(
        uint256 reserveId,
        uint256 eTokenAmount,
        address to,
        bool receiveNativeETH
    )
        public
        payable
        notPaused
        nonReentrant
        avoidUsingNativeEther
        returns (uint256)
```

```
    {...}
```

Listing 3.7: `LendingPool::redeem()`

Moreover, we observe there exists certain redundancy in unwrapping `WETH` back to the native `Ether` in a number of routines, including `closeVaultPositionPartially()`, `closeOutOfRangePosition()`, `liquidateVaultPositionPartially()`, `investEarnedFeeToLiquidity()`, and `exactRepay()`. The redundancy can be better optimized.

Recommendation Consider the removal of the redundant code with a simplified, consistent implementation.

Status The issue has been fixed by this commit: `ebed8b1`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Extra Finance` protocol, which is a community-driven leveraged yield farming (`LYF`) protocol and offers up to `3X` leverage. It enables users to farm a diverse range of farming pools on `Velodrome` and other `DEXes`. Users can customize their farming strategies with options like re-investing, market-neutral, and long/short farming strategies. In addition to `LYF`, `Extra Finance` also functions as a lending protocol. Users can deposit funds into its lending pools to earn interest on their deposited assets. This feature provides users with a way to earn passive income. The current code base is well organized and those identified issues are promptly confirmed and fixed.

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-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.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-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

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

[9] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.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.