SMART CONTRACT AUDIT REPORT

for

GoodEntry

Prepared By: Xiaomi Huang

PeckShield

May 25, 2023

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	May 25, 2023	Stephen Bie	Final Release
1.0-rc	May 15, 2023	Stephen Bie	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 GoodEntry 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 GoodEntry

GoodEntry is a non-custodial decentralized derivative exchange enabling leveraged day trading with built-in downside protection, built on top of Uniswap V3 and built-in lending market. Liquidity providers can earn swap and trading fees without counterparty risk. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of GoodEntry

Item	Description
Target	GoodEntry
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	May 25, 2023

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit. Note that the GoodEntry protocol assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit.

- https://github.com/GoodEntry-io/GoodEntryMarkets.git (2e3d230)

- https://github.com/ezoia-com/ge.git (d2846ba)

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

- https://github.com/GoodEntry-io/GoodEntryMarkets.git (2e3d230)
- https://github.com/ezoia-com/ge.git (54790b8)

1.2 About PeckShield

PeckShield Inc. [13] 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 [12]:

- 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) [11], 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 `GoodEntry` 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	3	■ ■ ■
Low	1	■
Informational	1	■
Undetermined	1	■
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 1 high-severity vulnerability, 3 medium-severity vulnerabilities, 1 low-severity vulnerability, 1 informational recommendation, and 1 undetermined issue.

Table 2.1: Key GoodEntry Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Lack of Health Factor Check in LonggPositionManager::clos()	Business Logic	Fixed
PVE-002	Undetermined	Possible Donation-Based Price Manipulation for UniswapV2 LP Market	Time and State	Confirmed
PVE-003	Medium	Revisited Logic of ZapBox::zapInSingleAsset()	Business Logic	Fixed
PVE-004	Medium	Revisited Liquidity Calculation in GeVault::deposit()	Business Logic	Fixed
PVE-005	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-006	Informational	Meaningful Events for Important State Changes	Coding Practices	Fixed
PVE-007	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 Lack of Health Factor Check in LonggPositionManager::clos()

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `LonggPositionManager`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In the `GoodEntry` protocol, the `LonggPositionManager` contract is designed to manage the user's `UniswapV2 LP` debt position. In particular, the `clos()` routine is designed to close the user's `UniswapV2 LP` debt position and then transfer the remaining collateral (i.e., `token0` and `token1` of the `UniswapV2 LP`) to him if needed. While examining its logic, we observe its current implementation needs to be improved.

To elaborate, we show below the related code snippet of the `LonggPositionManager` contract. Inside the `clos()` routine, it withdraws the user's collateral `param.assetA` (i.e., token0 of the `UniswapV2 LP`) and `param.assetB` (i.e., token1 of the `UniswapV2 LP`) from the lending market (lines 309 - 310). (Note that the `LendingPool::PMTransfer()` routine called inside the `PositionManager::PMWithdraw()` routine does not check the health factor according to the design.) Then part of them are exchanged to the user specified `debtAsset` (i.e., `UniswapV2 LP`) (line 315) to repay the user's debt (line 316). The remaining assets are swapped to the user specified `remainingAsset` (lines 319 - 326) and the swapped-out `remainingAsset` is transferred to the user without health factor validation. If the user borrows other assets (excluding the `debtAsset`) using the `param.assetA` and/or `param.assetB` as collateral, he may leave bad debts on the protocol. What's more, a malicious actor can exploit the vulnerability to steal the assets from the lending market.

```
300     function clos(uint poolId, address user, address debtAsset, uint repayAmount,
            address remainingAsset, bool withdrawCollateral) internal
```

```
    {
        DelevParam memory param;
        (param.lendingPool, param.oracle, param.ammRouter,,) = getPoolAddresses(poolId);
        param.debtAsset = debtAsset;
        param.repayAmount = repayAmount;
        (param.assetA, param.amtA, param.assetB, param.amtB) = prepareParams(address(
            param.lendingPool), param.debtAsset, user);

        // Transfer underlying tokens here + withdraw
        PMWithdraw(param.lendingPool, user, param.assetA, param.amtA);
        PMWithdraw(param.lendingPool, user, param.assetB, param.amtB);

        ...

        // Step 2: both tokens in sufficient quantity, addLiquidity and return LP
        transferAndMint(param.debtAsset, param.assetA, debtA, param.assetB, debtB, debt
            );
        param.lendingPool.repay( param.debtAsset, debt, 2, user);

        // Step 3: do we want to keep all of the tokens?
        if (msg.sender == user) {
            if ( remainingAsset == param.assetA ){
                swapAllTokens(param.ammRouter, param.assetB, param.assetA);
            }
            else if (remainingAsset == param.assetB ){
                swapAllTokens(param.ammRouter, param.assetA, param.assetB);
            }
        }

        // Step 4: send back funds to user
        if (withdrawCollateral){
            uint remaining = IERC20(remainingAsset).balanceOf(address(this));
            IERC20(remainingAsset).safeTransfer(user, remaining);
        }
        cleanup(param.lendingPool, user, param.assetA);
        cleanup(param.lendingPool, user, param.assetB);
        ...
    }
```

Listing 3.1: `LonggPositionManager::clos()`

```
    function PMTransfer(
        address aAsset,
        address user,
        uint256 amount
    ) external whenNotPaused {
        require(pm[msg.sender], "Not PM");
        if (tx.origin != user) {
            (,,,, uint256 healthFactor) = GenericLogic.calculateUserAccountData(
                user,
                _reserves,
                _usersConfig[user],
                _reservesList,
```

```
                _reservesCount,
                _addressesProvider.getPriceOracle()
                );
            require(healthFactor <= softLiquidationThreshold, "Not initiated by user");
        }
        IAToken(aAsset).transferOnLiquidation(user, msg.sender, amount);
    }
```

Listing 3.2: `LendingPool::PMTransfer()`

Recommendation Apply necessary sanity checks of the resulting health factor in the `clos()` routine.

Status The issue has been addressed in the following commit: `6045bc7`.

3.2 Possible Donation-Based Price Manipulation for UniswapV2 LP Market

- ID: PVE-002
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `LPOracle`
- Category: Time and State [10]
- CWE subcategory: CWE-682 [4]

Description

In the `GoodEntry` protocol, the lending market supports the `UniswapV2 LP` token. In particular, the fair `UniswapV2 LP` price is calculated according to the following formula: $P = 2 * \sqrt{r_0 * r1} * \sqrt{p_0 * p1}/totalSupply$. While examining the formula, we observe there is a donation-based price manipulation vulnerability that can be exploited to steal the assets from the lending market.

Using the `UNI-V2_WETH_USDC` as an example, assuming the collateral factor of `UNI-V2_WETH_USDC /USDC` is `0.8` and the total value of the `UNI-V2_WETH_USDC` pair is `$1M`, a malicious actor firstly supplies `$10M USDC` as collateral (Step I), secondly borrows `$8M UNI-V2_WETH_USDC` token via leverage (Step II), next donates `$2M USDC` to the `UNI-V2_WETH_USDC` pair to increase the `UNI-V2_WETH_USDC` token price (Step III) (That is to say, the new `UNI-V2_WETH_USDC` token price is three times the old one.), and finally borrows `$19.2M USDC` from the lending market with `$24M UNI-V2_WETH_USDC` token as collateral (Step IV). Overall, the malicious actor profits `$19.2M - $10M - $2M = $7.2M` leaving lending market with bad debt.

```
    function latestAnswer() external view returns (int256) {
        (uint a, uint b,) = LP_TOKEN.getReserves();

```

```
        uint priceA = uint(getAnswer(CL_TOKENA));
        uint priceB = uint(getAnswer(CL_TOKENB));

        ...

        uint norm_b;
        if (decimalsB >= decimalsA) {
            norm_b = sqrt( a * b * priceA * 10**(decimalsB-decimalsA) / priceB );
        } else {
            norm_b = sqrt( a * b * priceA / 10**(decimalsA-decimalsB) / priceB );
        }
        uint norm_a = a * b / norm_b;

        /*
            The normalised positions (18 decimals) are multiplied with the chainlink value (8 decimals), giving val.
            val is divided by LP_TOKEN.totalSupply(), which has 18 decimals, and casted to an int
            The return value represents the value * 10**8 of a single LP token
        */
        require(decimalsA <= 18 && decimalsB <= 18, "Incorrect tokens");
        uint val = norm_a * priceA * 10**(18-decimalsA) + norm_b * 10**(18-decimalsB) * priceB;
        return int(val / LP_TOKEN.totalSupply());
    }
```

Listing 3.3: `LPOracle::latestAnswer()`

Recommendation Revisit the `UniswapV2 LP` token support in the lending market.

Status The issue has been confirmed by the team. The team has been aware of such attack vector, and believe it can be prevented by limiting the amount of LP tokens that can be deposited in the lending pool (preventing leverage).

3.3 Revisited Logic of ZapBox::zapInSingleAsset()

- ID: PVE-003
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `ZapBox`
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [5]

Description

In the `ZapBox` contract, the `zapInSingleAsset()` routine is used by the user to exchange a kind of underlying token to the specified `UniswapV2 LP` token and deposit them into the lending market. While examining its logic, we observe its current implementation needs to be improved.

To elaborate, we show below the related code snippet of the `ZapBox` contract. Inside the `zapInSingleAsset()` routine, it swaps part of the incoming `token0` to `token1` via the call to `swapExactTokens-ForTokens()` (lines 150 - 156). Then the `addLiquidity()` routine is executed (line 159) to add the remaining `token0` and the swapped-out `token1` into the `UniswapV2` pool (specified by the input `token0` and `token1`). However, it comes to our attention that the input `amountADesired` parameter of the `addLiquidity()` routine is set to `swapped[0]` incorrectly, which is the used `token0` amount in the previous swap process rather than the remaining `token0` amount. Given this, we suggest to improve the implementation as below: `(amountA, amountB, liquidity)= router.addLiquidity(token0, token1, amount0 - swapped[0], swapped[1], (amount0 - swapped[0])*99/100 , swapped[1]*99/100, address(this), block .timestamp)` (line 159).

```
136     function zapInSingleAsset(uint poolId, address token0, uint amount0, address token1,
            uint amount1)
137         external
138         nonReentrant()
139         returns (uint amountA, uint amountB, uint liquidity)
140     {
141         require ( amount0 > 0, "ZERO_AMOUNT" );
142         (ILendingPool lp, IUniswapV2Router01 router) = getPoolAddresses(poolId);
143         address lpToken = IUniswapV2Factory(router.factory()).getPair(token0, token1);
144         ERC20(token0).transferFrom(msg.sender, address(this), amount0);
145 
146         address[] memory path = new address[](2);
147         path[0] = token0; path[1] = token1;
148 
149         checkSetApprove(token0, address(router), amount0);
150         uint[] memory swapped = router.swapExactTokensForTokens(
151             getSwapAmt(lpToken, token0, amount0),
152             amount1,
153             path,
154             address(this),
155             block.timestamp
156         );
157         checkSetApprove(token1, address(router), swapped[1]);
158 
159         (amountA, amountB, liquidity) = router.addLiquidity(token0, token1, swapped[0],
                swapped[1], swapped[0]*99/100 , swapped[1]*99/100, address(this), block.
                timestamp);
160 
161         ...
162     }
```

Listing 3.4: `ZapBox::zapInSingleAsset()`

Recommendation Improve the implementation of the `zapInSingleAsset()` routine as above-mentioned.

Status The issue has been addressed in the following commit: `21368bb`.

3.4 Revisited Liquidity Calculation in GeVault::deposit()

- ID: PVE-004
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `GeVault`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In the `GoodEntry` protocol, the `GeVault` contract is one of the main entries for interaction with users. In particular, one entry routine, i.e., `deposit()`, allows the user to deposit the supported `token0` and `token1` and get in return LP token to represent the vault shares. While examining its logic, we observe its current implementation needs to be improved.

To elaborate, we show below the related code snippet of the `GeVault` contract. Inside the `deposit()` routine, we notice part of the incoming assets are transferred to the `treasury` (lines 229 - 230) as deposit fee. After further analysis, we observe the value (i.e., `valueX8`, line 226) of all the incoming assets (including the deposit fee) is used to calculate the share amount (line 240), which directly undermines the assumption of the design.

```
222     function deposit(address token, uint amount) public payable nonReentrant returns (uint liquidity)
223     {
224         ...
225 
226         uint valueX8 = oracle.getAssetPrice(token) * amount / 10**ERC20(token).decimals();
227         require(tvlCap > valueX8 + getTVL(), "GEV: Max Cap Reached");
228         // Send deposit fee to treasury
229         uint fee = amount * getAdjustedBaseFee(token == address(token0)) / 1e4;
230         ERC20(token).safeTransfer(treasury, fee);
231         uint splitAmount = (amount - fee)/2;
232         uint vaultValueX8 = getTVL();
233 
234         uint tSupply = totalSupply();
235         // initial liquidity at 1e18 token ~ $1
236         if (tSupply == 0)
237             liquidity = valueX8 * 1e10;
238         else {
239             liquidity = valueX8 * 1e10 * tSupply;
240             if (vaultValueX8 > 0) liquidity = tSupply * valueX8 / vaultValueX8;
241         }
242         ...
243     }
```

Listing 3.5: `GeVault::deposit()`

Recommendation Properly exclude the deposit fee during calculating the share amount.

Status The issue has been addressed in the following commit: `21368bb`.

3.5 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1109 [1]

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 token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }
    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
```

```
    }
```

Listing 3.6: `ZRX.sol`

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. Similarly, there is a safe version of `transferFrom()/approve()` as well, i.e., `safeTransferFrom()/safeApprove()`.

In the following, we show the `TokenisableRange::claimFee()` routine. If the `USDT` token is supported as `TOKEN0.token`, the unsafe version of `TOKEN0.token.transfer(TREASURY, tf0)` (line 174) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value). We may intend to replace `TOKEN0.token.transfer(TREASURY, tf0)` (line 174) with `safeTransfer()`.

```
161     function claimFee() public {
162         (uint256 newFee0, uint256 newFee1) = POS_MGR.collect(
163         INonfungiblePositionManager.CollectParams({
164             tokenId: tokenId,
165             recipient: address(this),
166             amount0Max: type(uint128).max,
167             amount1Max: type(uint128).max
168         })
169         );
170         // If there's no new fees generated, skip compounding logic;
171         if ((newFee0 == 0) && (newFee1 == 0)) return;
172         uint tf0 = newFee0 * treasuryFee / 100;
173         uint tf1 = newFee1 * treasuryFee / 100;
174         if (tf0 > 0) TOKEN0.token.transfer(TREASURY, tf0);
175         if (tf1 > 0) TOKEN1.token.transfer(TREASURY, tf1);
176 
177         ...
178     }
```

Listing 3.7: `TokenisableRange::claimFee()`

Note that this issue is present in a number of contracts, including `LonggPositionManager`, `Position-Manager`, `GeVault`, `RangeManager`, `TokenisableRange`, `ZapBox`, and `ZapBoxTR`.

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()`, `transferFrom()`, and `approve()`. And there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Status The issue has been addressed in the following commit: `21368bb`.

3.6 Meaningful Events for Important State Changes

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `GeVault/HardcodedPriceOracle`
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [3]

Description

The `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

While examining the events that reflect the protocol dynamics, we notice there are several privileged routines that lack meaningful events to reflect their changes. In the following, we show several representative routines.

```
    /// @notice Set pool status
    /// @param _isEnabled Pool status
    function setEnabled(bool _isEnabled) public onlyOwner { isEnabled = _isEnabled; }

    /// @notice Set treasury address
    /// @param newTreasury New address
    function setTreasury(address newTreasury) public onlyOwner { treasury = newTreasury;
        }
```

Listing 3.8: `GeVault`

With that, we suggest to emit meaningful events in these privileged routines. Also, the key event information is better `indexed`. Note each emitted event is represented as a topic that usually consists of the signature (from a `keccak256` hash) of the event name and the types (`uint256`, `string`, etc.) of its parameters. Each indexed type will be treated like an additional topic. If an argument is not indexed, it will be attached as data (instead of a separate topic). Considering that the key information is typically queried, it is better treated as a topic, hence the need of being `indexed`.

Recommendation Properly emit the above-mentioned events with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status The issue has been addressed in the following commit: `21368bb`.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [7]
- CWE subcategory: CWE-287 [2]

Description

In the GoodEntry protocol, there is a privileged owner account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and price oracle adjustment). Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
    function setAssetSources(address[] calldata assets, address[] calldata sources)
    external
    onlyOwner
    {
        _setAssetsSources(assets, sources);
    }

    function setFallbackOracle(address fallbackOracle) external onlyOwner {
        _setFallbackOracle(fallbackOracle);
    }
```

Listing 3.9: AaveOracle

```
    function setHardcodedPrice(int256 _hardcodedPrice) external onlyOwner {
        hardcodedPrice = _hardcodedPrice;
    }
```

Listing 3.10: HardcodedPriceOracle

```
    /// @notice Set pool status
    /// @param _isEnabled Pool status
    function setEnabled(bool _isEnabled) public onlyOwner { isEnabled = _isEnabled; }

    /// @notice Set treasury address
    /// @param newTreasury New address
    function setTreasury(address newTreasury) public onlyOwner { treasury = newTreasury;
         }
```

Listing 3.11: GeVault

Moreover, the LendingPoolAddressesProvider contract allows the privileged owner to configure protocol-wide contracts, including LENDING_POOL, LENDING_POOL_CONFIGURATOR, POOL_ADMIN, EMERGENCY_ADMIN , LENDING_POOL_COLLATERAL_MANAGER, PRICE_ORACLE, and LENDING_RATE_ORACLE. These contracts play a variety of duties and are also considered privileged.

```
contract LendingPoolAddressesProvider is Ownable, ILendingPoolAddressesProvider {
    string private _marketId;
    mapping(bytes32 => address) private _addresses;

    bytes32 private constant LENDING_POOL = 'LENDING_POOL';
    bytes32 private constant LENDING_POOL_CONFIGURATOR = 'LENDING_POOL_CONFIGURATOR'
        ;
    bytes32 private constant POOL_ADMIN = 'POOL_ADMIN';
    bytes32 private constant EMERGENCY_ADMIN = 'EMERGENCY_ADMIN';
    bytes32 private constant LENDING_POOL_COLLATERAL_MANAGER = 'COLLATERAL_MANAGER';
    bytes32 private constant PRICE_ORACLE = 'PRICE_ORACLE';
    bytes32 private constant LENDING_RATE_ORACLE = 'LENDING_RATE_ORACLE';
    ...
}
```

Listing 3.12: `LendingPoolAddressesProvider`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is 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. For the time being, it is planned to mitigate with a timelock mechanism.

4 Conclusion

In this audit, we have analyzed the design and implementation of `GoodEntry`, which is a non-custodial decentralized derivative exchange enabling leveraged day trading with built-in downside protection, built on top of `Uniswap V3` and built-in lending market. It allows liquidity providers to earn swap and trading fees without counterparty risk. 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-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[5] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[6] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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

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

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

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