SMART CONTRACT AUDIT REPORT

for

GoodEntry (v2)

Prepared By: Xiaomi Huang

PeckShield

December 26, 2023

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	December 26, 2023	Xuxian Jiang	Final Release
1.0-rc	December 21, 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 GoodEntry (v2) 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. It also seamlessly integrates with Uniswap V2, Uniswap V3, and Camelot V3 for increased capital efficiency. 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	December 26, 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/ge-v2.git (ff79667)

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/ge-v2.git (304c86c)

1.2 About PeckShield

PeckShield Inc. [11] 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 [10]:

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

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

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

Table 1.3: The Full List of Check Items

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

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

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

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], 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 `GoodEntry (v2)` 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	4	■ ■ ■ ■
Informational	0	
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 2 medium-severity vulnerabilities and 4 low-severity vulnerabilities.

Table 2.1: Key GoodEntry Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Anti-Flashloan Effectiveness in GoodEntryVaultBase	Time And State	Resolved
PVE-002	Medium	Improper Fee Accounting in GoodEntryVaultBase	Time and State	Resolved
PVE-003	Low	Incorrect tokenURI() Generation in GoodEntryPositionManager	Business Logic	Resolved
PVE-004	Low	Improved Gas Efficiency in UniswapV3Position	Coding Practices	Resolved
PVE-005	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-006	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 Anti-Flashloan Effectiveness in GoodEntryVaultBase

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: GoodEntryVaultBase
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

In the GoodEntry (v2) protocol, there is a protection mechanism that is designed to counter flashloan attacks. The protection mechanism works as follows: it marks the user deposit time and ensures the user withdrawal may not occur in the same block with the deposit. While examining this protection mechanism, we observe it may be bypassed.

To elaborate, we show below the related withdrawal logic implemented in the _withdraw() routine. It performs a number of essential validation checks to ensure the user may be allowed to withdraw. Specifically, the check (line 80) ensures the withdraw will not occur on the same block with the previous deposit. Note the user's depositTime[user] is updated upon the user deposit. However, our analysis shows that the above checks may be bypassed.

```
74  function _withdraw(address user, uint liquidity, address token) internal nonReentrant
        returns (uint amount){
75    require(token == address(baseToken)  token == address(quoteToken), "GEV: Invalid
        Token");
76    require(liquidity <= balanceOf(user), "GEV: Insufficient Balance");
77    if(liquidity == 0) liquidity = balanceOf(user);
78    if(liquidity == 0) return 0;
79
80    require(block.timestamp > depositTime[user] && liquidity <= depositBalance[user], "
        GEV: Early Withdrawal");
81    depositBalance[user] -= liquidity; // could updateUserBlance but maybe user wants 2
        withdrawals in same block, eg half base half quote
82
```

```
  (,,uint vaultValueX8) = getReserves();
  uint valueX8 = vaultValueX8 * liquidity / totalSupply();
  amount = valueX8 * 10**ERC20(token).decimals() / oracle.getAssetPrice(token);
  uint fee = amount * getAdjustedBaseFee(token == address(quoteToken)) /
      FEE_MULTIPLIER;

  _burn(user, liquidity);
  withdrawAmm();
  ...
}
```

Listing 3.1: `GoodEntryVaultBase::_withdraw()`

```
/// @notice Update a user balance and deposit time
/// @dev This prevents flashloan attacks, as a user cannot deposit+withdraw in the
    same block, and cannot withdraw funds received through token transfer
function updateUserBalance() public {
  depositTime[msg.sender] = block.timestamp;
  depositBalance[msg.sender] = balanceOf(msg.sender);
}
```

Listing 3.2: `GoodEntryVaultBase::updateUserBalance()`

In particular, a user `Malice` may make a deposit (with the amount `amt`) so that `Malice`'s `depositTime` and `depositBalance` are accordingly updated. `Malice` can then transfer the vault share token to another address `A0` and call `updateUserBalance()` to update `A0`'s `depositTime` and `depositBalance`. In fact, `Malice` can repeat the process to create `N` addresses so that `N*amt` will be equal to the flashloan amount. Next, when the flashloan deposit is made, `Malice` can transfer each `amt` to each above address and perform the withdraw from that address without being blocked.

Recommendation Revisit the above countermeasure to make it flashloan-resistant.

Status The issue has been addressed in the following commit: `127c6f6`.

3.2 Improper Fee Accounting in GoodEntryVaultBase

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: `GoodEntryVaultBase`
- Category: Business Logic [7]
- CWE subcategory: CWE-837 [4]

Description

The `GoodEntry (v2)` protocol has a `FeeStreamer` contract that accounts for the collected fees in a given period for later distribution. While analyzing the fee collection logic, we notice the repayment logic of borrowed fund does not properly account for the accumulated fee.

In the folowing, we show the implementation of the related `repay()` routine. While it properly collects the payment in the contract and sends out the treasury share, we notice the accounted-for fee reserve amount should be `fees * (100-goodEntryCore.treasuryShareX2())/ 100`, not `fees` (line 216).

```
204  function repay(address token, uint amount, uint fees) public onlyOPM nonReentrant {
205    require(amount > 0, "GEV: Invalid Debt");
206    require(poolPriceMatchesOracle(), "GEV: Oracle Error");
207    withdrawAmm();
208
209    if(token == address(quoteToken)) quoteToken.safeTransferFrom(msg.sender, address(
           this), amount + fees);
210    else {
211      ERC20(token).safeTransferFrom(msg.sender, address(this), amount);
212      quoteToken.safeTransferFrom(msg.sender, address(this), fees);
213    }
214    oracle.getAssetPrice(address(quoteToken));
215    if (fees > 0) {
216      reserveFees(0, fees, fees * oracle.getAssetPrice(address(quoteToken)) / 10**
           quoteToken.decimals());
217      quoteToken.safeTransfer(goodEntryCore.treasury(), fees * goodEntryCore.
           treasuryShareX2() / 100);
218    }
219    deployAssets();
220    emit Repaid(token, amount);
221  }
```

Listing 3.3: `GoodEntryVaultBase::repay()`

Recommendation Revise the above routine to properly computed the accounted-for reserve fee amount.

Status The issue has been addressed in the following commit: `84059a3`.

3.3 Incorrect tokenURI() Generation in GoodEntryPositionManager

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: GoodEntryPositionManager
- Category: Coding Practices [6]
- CWE subcategory: CWE-1109 [1]

Description

Each position in GoodEntry (v2) is represented as an NFT, which can be computed via the tokenURI() routine. Our analysis on this routine shows it has an issue in mixing up the appearance order between baseToken and quoteToken.

To elaborate, we show below the related code snippet of the tokenURI() routine. The purpose here is to provide a nice NFT representation of a position. While it makes use of an external contract GENFT_PROXY for its constructTokenURI() function, we notice this function has the ConstructTokenURIParams input that has seven fields. And the fourth and fifth member fields should be quoteTokenSymbol and baseTokenSymbol. However, current implementation (line 106) mixes up these two fields.

```
101    function tokenURI(uint256 tokenId) public view override(ERC721, IERC721Metadata)
           returns (string memory) {
102      Position memory position = _positions[tokenId];
103      (, uint pnl) = getValueAtStrike(position.isCall, IGoodEntryVault(vault).getBasePrice
             (), position.strike, position.notionalAmount);
104      int actualPnl = int(pnl) - int(getFeesAccumulated(tokenId));
105      return IGeNftDescriptor(GENFT_PROXY).constructTokenURI(IGeNftDescriptor.
             ConstructTokenURIParams(
106        tokenId, address(quoteToken), address(baseToken), baseToken.symbol(), quoteToken.
               symbol(), position.isCall, actualPnl
107      ));
108
109    }
110
111  struct ConstructTokenURIParams {
112      uint256 tokenId;
113      address quoteTokenAddress;
114      address baseTokenAddress;
115      string quoteTokenSymbol;
116      string baseTokenSymbol;
117      bool isCall;
118      int pnl;
119    }
```

Listing 3.4: GoodEntryPositionManager::tokenURI()

Recommendation Revise the above routine by using the correct order of member fields in ConstructTokenURIParams.

Status The issue has been addressed in the following commit: 8e17dec.

3.4 Improved Gas Efficiency in UniswapV3Position

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: UniswapV3Position
- Category: Coding Practices [6]
- CWE subcategory: CWE-1109 [1]

Description

As mentioned earlier, the GoodEntry (v2) protocol seamlessly integrates with Uniswap V2, Uniswap V3 , and Camelot V3. While reviewing the Uniswap V3 integration, we notice possible improvement in optimizing gas usage

To elaborate, we show below the related code snippet of the _getReserves() routine. This routine is designed to report current reserve amount managed by the Uniswap V3 position. As this routine is often used, the internal repeated calculation of TickMath.getSqrtRatioAtTick (line 159) may be avoided by caching the computation when the tick range is chosen.

```
154  function _getReserves() internal override view returns (uint baseAmount, uint
         quoteAmount){
155    uint token0Amount; uint token1Amount;
156    uint128 liquidity = uint128(getLiquidity());
157    if (liquidity > 0){
158      uint160 sqrtPriceX96 = _sqrtPriceX96();
159      (token0Amount, token1Amount) = LiquidityAmounts.getAmountsForLiquidity(
             sqrtPriceX96, TickMath.getSqrtRatioAtTick(lowerTick), TickMath.
             getSqrtRatioAtTick(upperTick),  liquidity);
160    }
161    (baseAmount, quoteAmount) = baseToken < quoteToken ? (token0Amount, token1Amount) :
           (token1Amount, token0Amount);
162  }
```

Listing 3.5: UniswapV3Position::_getReserves()

Recommendation Cache the first calculation and later use the cached result instead of making repeated computation.

Status The issue has been addressed in the following commit: 9fc7472.

3.5 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- 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 `GoodEntryVaultBase::borrow()` routine. If the `USDT` token is supported as `token`, the unsafe version of `ERC20(token).transfer(msg.sender, amount)` (line 197) 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 it with `safeTransfer()`.

```
192   function borrow(address token, uint amount) public onlyOPM nonReentrant {
193     require(!goodEntryCore.isPaused(), "GEV: Pool Disabled");
194     withdrawAmm();
195     require(ERC20(token).balanceOf(address(this)) >= amount, "GEV: Not Enough Supply");
196 
197     ERC20(token).transfer(msg.sender, amount);
198     deployAssets();
199     emit Borrowed(address(token), amount);
200   }
```

Listing 3.7: `GoodEntryVaultBase::borrow()`

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()`, `transferFrom()`, and `approve()`.

Status The issue has been addressed in the following commit: `ededa23`.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `GoodEntry (v2)` 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 setPaused(bool _isPaused) public onlyRole(PAUSER_ROLE) {
    isPaused = _isPaused;
    emit SetPaused(_isPaused);
  }


  /// @notice Set treasury address
  /// @param _treasury New address
  function setTreasury(address _treasury, uint8 _treasuryShareX2) public onlyRole(DEFAULT_ADMIN_ROLE) {
    require(_treasury != address(0x0), "GEC: Invalid Treasury");
    require(_treasuryShareX2 <= 100, "GEC: Invalid Treasury Share");
    treasury = _treasury;
    treasuryShareX2 = _treasuryShareX2;
    emit SetTreasury(_treasury, _treasuryShareX2);
  }


  /// @notice Set treasury address
  function setPermissionlessVaultCreation(bool _isPermissionlessVaultCreation) public onlyRole(DEFAULT_ADMIN_ROLE) {
    isPermissionlessVaultCreation = _isPermissionlessVaultCreation;
    emit SetPermissionlessVaultCreation(_isPermissionlessVaultCreation);
  }


  /// @notice Add a new GoodEntry Vault proxy, e.g supporting a new AMM
  function setVaultUpgradeableBeacon(address _vaultUpgradeableBeacon, bool isEnabled) public onlyRole(DEFAULT_ADMIN_ROLE) {
    vaultUpgradeableBeacons[_vaultUpgradeableBeacon] = isEnabled;
    vaultImplementations[IGoodEntryVault(payable(UpgradeableBeacon(_vaultUpgradeableBeacon).implementation())).ammType()] = _vaultUpgradeableBeacon;
    emit SetVaultUpgradeableBeacon(_vaultUpgradeableBeacon, isEnabled);
```

```
}


/// @notice Upgrade the TokenisableRange implementations
function updateVaultBeacon(address _vaultUpgradeableBeacon, address _newVaultImpl)
    public onlyRole(DEFAULT_ADMIN_ROLE) {
  require(_newVaultImpl != address(0x0), "GEC: Invalid Beacon");
  require(
    keccak256(abi.encodePacked(IGoodEntryVault(payable(UpgradeableBeacon(
        _vaultUpgradeableBeacon).implementation())).ammType()))
    == keccak256(abi.encodePacked(IGoodEntryVault(payable(_newVaultImpl)).ammType() ))
        ,
    "GEC: Wrong Impl");
  UpgradeableBeacon(_vaultUpgradeableBeacon).upgradeTo(_newVaultImpl);
  emit SetVaultBeacon(_vaultUpgradeableBeacon, _newVaultImpl);
}
```

Listing 3.8: Privileged Operations in GoodEntryCore

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 the `GoodEntry (v2)` protocol, which is a non-custodial decentralized derivative exchange. It enables leveraged day trading with built-in downside protection. It also seamlessly integrates with `Uniswap V2`, `Uniswap V3`, and `Camelot V3` for increased capital efficiency. 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-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[4] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.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 VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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