SMART CONTRACT AUDIT REPORT

for

Revert V3utils

Prepared By: Xiaomi Huang

PeckShield

March 16, 2023

Document Properties

Client	Revert Finance
Title	Smart Contract Audit Report
Target	Revert V3utils
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 16, 2023	Xiaotao Wu	Final Release
1.0-rc	March 5, 2023	Xiaotao Wu	Release Candidate #1

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 V3utils feature, 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 Revert V3utils

The V3utils of Revert Finance is a set of convenience functions for Uniswap LPs allowing them to perform tasks that would require multiple transactions if interacting with the NFTManager contract, in one atomic transaction. It is intended to be used alongside the Revert front-end and a swap aggregator like the 0x protocol. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Revert V3utils

Item	Description
Name	Revert Finance
Website	https://revert.finance//
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 16, 2023

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

- https://github.com/revert-finance/v3utils.git (336453d)

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

- https://github.com/revert-finance/v3utils.git (fb7d37d)

1.2 About PeckShield

PeckShield Inc. [9] 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 [8]:

- 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) [7], 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 design and implementation of the `V3utils` feature. 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 logics, 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	0	
Low	2	■ ■
Informational	1	■
Total	3	

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 low-severity vulnerabilities and 1 informational suggestion.

Table 2.1: Key Revert V3utils Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Lack of Slippage Control in onERC721Received()	Time and State	Fixed
PVE-002	Informational	Incorrect swapSourceToken Using in onERC721Received()	Business Logic	Fixed
PVE-003	Low	Accommodation Of Non-ERC20-Compliant Tokens	Coding Practices	Fixed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Lack of Slippage Control in onERC721Received()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: V3Utils
- Category: Time and State [4]
- CWE subcategory: CWE-362 [2]

Description

The onERC721Received function in the V3Utils contract is designed to facilitate the user operations with UniswapV3 in collecting fees, decreasing/increasing liquidity, mint operations, etc. In particular, the internal _swapAndIncrease()/_swapAndMint() routines will be called to swap between the positions' tokens and add liquidity in UniswapV3.

While examining the input arguments of these two routines, we observe that there is no slippage control in place, which opens up the possibility for front-running and potentially results in a smaller LP amount. Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and a smaller return as expected to the liquidity provider. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

```
    function onERC721Received(address, address from, uint256 tokenId, bytes calldata
        data) external override returns (bytes4) {
        ...

        uint256 amount0;
        uint256 amount1;
        if (instructions.liquidity != 0) {
```

```
            (amount0, amount1) = _decreaseLiquidity(tokenId, instructions.liquidity,
                instructions.deadline, instructions.amountIn0, instructions.amountIn1);
        }
        (amount0, amount1) = _collectFees(tokenId, IERC20(token0), IERC20(token1),
            instructions.feeAmount0 == type(uint128).max ? type(uint128).max : (amount0
            + instructions.feeAmount0).toUint128(), instructions.feeAmount1 == type(
            uint128).max ? type(uint128).max : (amount1 + instructions.feeAmount1).
            toUint128());

        if (instructions.whatToDo == WhatToDo.COMPOUND_FEES) {
            if (instructions.targetToken == token0) {
                if (amount1 < instructions.amountIn1) {
                    revert AmountError();
                }
                (liquidity, amount0, amount1) = _swapAndIncrease(
                    SwapAndIncreaseLiquidityParams(tokenId, amount0, amount1,
                    instructions.recipient, instructions.deadline, IERC20(token1),
                    instructions.amountIn1, instructions.amountOut1Min, instructions.
                    swapData1, 0, 0, "", 0, 0), IERC20(token0), IERC20(token1),
                    instructions.unwrap);
            } else if (instructions.targetToken == token1) {
                if (amount0 < instructions.amountIn0) {
                    revert AmountError();
                }
                (liquidity, amount0, amount1) = _swapAndIncrease(
                    SwapAndIncreaseLiquidityParams(tokenId, amount0, amount1,
                    instructions.recipient, instructions.deadline, IERC20(token0), 0, 0,
                     "", instructions.amountIn0, instructions.amountOut0Min,
                    instructions.swapData0, 0, 0), IERC20(token0), IERC20(token1),
                    instructions.unwrap);
            } else {
                // no swap is done here
                (liquidity,amount0, amount1) = _swapAndIncrease(
                    SwapAndIncreaseLiquidityParams(tokenId, amount0, amount1,
                    instructions.recipient, instructions.deadline, IERC20(address(0)),
                    0, 0, "", 0, 0, "", 0, 0), IERC20(token0), IERC20(token1),
                    instructions.unwrap);
            }
            emit CompoundFees(tokenId, liquidity, amount0, amount1);
        } else if (instructions.whatToDo == WhatToDo.CHANGE_RANGE) {

            uint256 newTokenId;

            if (instructions.targetToken == token0) {
                if (amount1 < instructions.amountIn1) {
                    revert AmountError();
                }
                (newTokenId,,,) = _swapAndMint(SwapAndMintParams(IERC20(token0), IERC20(
                    token1), instructions.fee, instructions.tickLower, instructions.
                    tickUpper, amount0, amount1, instructions.recipient, from,
                    instructions.deadline, IERC20(token1), instructions.amountIn1,
                    instructions.amountOut1Min, instructions.swapData1, 0, 0, "", 0, 0,
```

```
                    instructions.swapAndMintReturnData), instructions.unwrap);
            } else if (instructions.targetToken == token1) {
                if (amount0 < instructions.amountIn0) {
                    revert AmountError();
                }
                (newTokenId,,,) = _swapAndMint(SwapAndMintParams(IERC20(token0), IERC20(
                    token1), instructions.fee, instructions.tickLower, instructions.
                    tickUpper, amount0, amount1, instructions.recipient, from,
                    instructions.deadline, IERC20(token0), 0, 0, "", instructions.
                    amountIn0, instructions.amountOut0Min, instructions.swapData0, 0, 0,
                     instructions.swapAndMintReturnData), instructions.unwrap);
            } else {
                // no swap is done here
                (newTokenId,,,) = _swapAndMint(SwapAndMintParams(IERC20(token0), IERC20(
                    token1), instructions.fee, instructions.tickLower, instructions.
                    tickUpper, amount0, amount1, instructions.recipient, from,
                    instructions.deadline, IERC20(token0), 0, 0, "", 0, 0, "", 0, 0,
                    instructions.swapAndMintReturnData), instructions.unwrap);
            }

            emit ChangeRange(tokenId, newTokenId);
        } else if (instructions.whatToDo == WhatToDo.WITHDRAW_AND_COLLECT_AND_SWAP) {
            ...
        } else {
            revert NotSupportedWhatToDo();
        }

        // return token to owner (this line guarantees that token is returned to
            originating owner)
        nonfungiblePositionManager.safeTransferFrom(address(this), from, tokenId,
            instructions.returnData);

        return IERC721Receiver.onERC721Received.selector;
    }
```

Listing 3.1: V3Utils::onERC721Received()

Furthermore, the slippage check for decreasing liquidity and collecting fees in current implementation can be improved. If `instructions.liquidity != 0`, the current slippage check will always be true because the `_decreaseLiquidity()` function requires `(amount0 >= instructions.amountIn0 && amount1 >= instructions.amountIn0`.

Recommendation Develop an effective mitigation to the above sandwich arbitrage to better protect the interests of users.

Status This issue has been fixed in the following commits: `9176c99`. `fb7d37d`.

3.2 Incorrect swapSourceToken Using in onERC721Received()

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `V3Utils`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned in Section 3.1, the `onERC721Received` function in the `V3Utils` contract is designed to facilitate the user operations with `UniswapV3`. While examining the logics of this routine, we notice the input arguments for the internal `_swapAndMint()` routine is not correctly passed.

To elaborate, we show below the related code snippet. It comes to our attention that the input argument `SwapAndMintParams.swapSourceToken` should be `IERC20(address(0))` if no swap operation is required, instead of current `IERC20(token0)` (line 157). Note this inappropriate input argument use will not change the execution result of the `_swapAndMint()` routine since the `token0` amount to be swapped is set to 0, but will consume more gas.

```
136     function onERC721Received(address, address from, uint256 tokenId, bytes calldata data) external override returns (bytes4) {
137         ...
138 
139         if (instructions.whatToDo == WhatToDo.COMPOUND_FEES) {
140             ...
141         } else if (instructions.whatToDo == WhatToDo.CHANGE_RANGE) {
142 
143             uint256 newTokenId;
144 
145             if (instructions.targetToken == token0) {
146                 if (amount1 < instructions.amountIn1) {
147                     revert AmountError();
148                 }
149                 (newTokenId,,,) = _swapAndMint(SwapAndMintParams(IERC20(token0), IERC20(token1), instructions.fee, instructions.tickLower, instructions.tickUpper, amount0, amount1, instructions.recipient, from, instructions.deadline, IERC20(token1), instructions.amountIn1, instructions.amountOut1Min, instructions.swapData1, 0, 0, "", 0, 0, instructions.swapAndMintReturnData), instructions.unwrap);
150             } else if (instructions.targetToken == token1) {
151                 if (amount0 < instructions.amountIn0) {
152                     revert AmountError();
153                 }
154                 (newTokenId,,,) = _swapAndMint(SwapAndMintParams(IERC20(token0), IERC20(token1), instructions.fee, instructions.tickLower, instructions.tickUpper, amount0, amount1, instructions.recipient, from, instructions.deadline, IERC20(token0), 0, 0, "", instructions.
```

```
                    amountIn0, instructions.amountOut0Min, instructions.swapData0, 0, 0,
                     instructions.swapAndMintReturnData), instructions.unwrap);
            } else {
                // no swap is done here
                (newTokenId,,,) = _swapAndMint(SwapAndMintParams(IERC20(token0), IERC20(
                    token1), instructions.fee, instructions.tickLower, instructions.
                    tickUpper, amount0, amount1, instructions.recipient, from,
                    instructions.deadline, IERC20(token0), 0, 0, "", 0, 0, "", 0, 0,
                    instructions.swapAndMintReturnData), instructions.unwrap);
            }

            emit ChangeRange(tokenId, newTokenId);
        } else if (instructions.whatToDo == WhatToDo.WITHDRAW_AND_COLLECT_AND_SWAP) {
            ...
        } else {
            revert NotSupportedWhatToDo();
        }

        // return token to owner (this line guarantees that token is returned to
            originating owner)
        nonfungiblePositionManager.safeTransferFrom(address(this), from, tokenId,
            instructions.returnData);

        return IERC721Receiver.onERC721Received.selector;
    }
```

Listing 3.2: V3Utils::onERC721Received()

Recommendation Pass the correct value to the input argument `SwapAndMintParams.swapSourceToken` of the `_swapAndMint()` routine.

Status This issue has been fixed in the following commit: `9176c99`.

3.3 Accommodation Of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `V3Utils`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [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 `approve()` routine and analyze 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. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()`/ `transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.3: `USDT Token Contract`

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Moreover, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the `approve()` function does not have a return value. However, the `IERC20` interface has defined the following `approve()` interface with a `bool` return value: `function approve(address spender, uint256 amount)external returns (bool)`. As a result, the call to `approve()` may expect a return value. With the lack of return value of `USDT`'s `approve()`, the call will be unfortunately reverted.

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`, 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 the following, we use the `_swapAndPrepareAmounts()` routine as an example. If the `USDT` token is supported as `params.token0` or `params.token1`, the unsafe version of `params.token0.approve()` and `params.token1.approve()` may return false but the current implementation fails to check the return value!

```
    function _swapAndPrepareAmounts(SwapAndMintParams memory params, bool unwrap) internal returns (uint256 total0, uint256 total1) {
        if (params.swapSourceToken == params.token0) {
            if (params.amount0 < params.amountIn1) {
                revert AmountError();
            }
            (uint256 amountInDelta, uint256 amountOutDelta) = _swap(params.token0, params.token1, params.amountIn1, params.amountOut1Min, params.swapData1);
            total0 = params.amount0 - amountInDelta;
            total1 = params.amount1 + amountOutDelta;
        } else if (params.swapSourceToken == params.token1) {
            if (params.amount1 < params.amountIn0) {
                revert AmountError();
            }
            (uint256 amountInDelta, uint256 amountOutDelta) = _swap(params.token1, params.token0, params.amountIn0, params.amountOut0Min, params.swapData0);
            total1 = params.amount1 - amountInDelta;
            total0 = params.amount0 + amountOutDelta;
        } else if (address(params.swapSourceToken) != address(0)) {

            (uint256 amountInDelta0, uint256 amountOutDelta0) = _swap(params.swapSourceToken, params.token0, params.amountIn0, params.amountOut0Min, params.swapData0);
            (uint256 amountInDelta1, uint256 amountOutDelta1) = _swap(params.swapSourceToken, params.token1, params.amountIn1, params.amountOut1Min, params.swapData1);
            total0 = params.amount0 + amountOutDelta0;
            total1 = params.amount1 + amountOutDelta1;

            // return third token leftover if any
            uint256 leftOver = params.amountIn0 + params.amountIn1 - amountInDelta0 - amountInDelta1;

            if (leftOver != 0) {
                _transferToken(params.recipient, params.swapSourceToken, leftOver, unwrap);
            }
        } else {
            total0 = params.amount0;
            total1 = params.amount1;
        }

        if (total0 != 0) {
            params.token0.approve(address(nonfungiblePositionManager), total0);
        }
        if (total1 != 0) {
            params.token1.approve(address(nonfungiblePositionManager), total1);
        }
    }
```

Listing 3.4: `V3Utils::_swapAndPrepareAmounts()`

Note similar issue also exists in the `_swap()` routine of the same contract.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`.

Status This issue has been fixed in the following commit: `9176c99`.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `V3utils` support in `Revert`. The `V3utils` of `Revert Finance` is a set of convenience functions for `Uniswap LPs` allowing them to perform tasks that would require multiple transactions if interacting with the `NFTManager` contract, in one atomic transaction. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based 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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

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

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

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

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

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

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

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