SMART CONTRACT AUDIT REPORT

for

Revert V4Utils

Prepared By: Xiaomi Huang

PeckShield

November 18, 2025

Document Properties

Client	Revert Finance
Title	Smart Contract Audit Report
Target	Revert V4Utils
Version	1.0
Author	Xuxian Jiang
Auditors	Matthew Jiang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 18, 2025	Xuxian Jiang	Final Release
1.0-rc	November 17, 2025	Xuxian Jiang	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 new V4Utils module in Revert, 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

Revert provides streamlined tools to manage UniswapV3/V4 LP positions. The LP positions may further serve as collateral and allow users to leverage by taking loans while retaining control and management of their capital within the UniswapV3/V4 pools. This audit covers the V4Utils contract to support UniswapV4 LP positions. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Revert V4Utils

Item	Description
Name	Revert Finance
Website	https://revert.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 18, 2025

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this audit only covers the new V4Utils contract.

- https://github.com/revert-finance/v4utils.git (ef5d6ff)

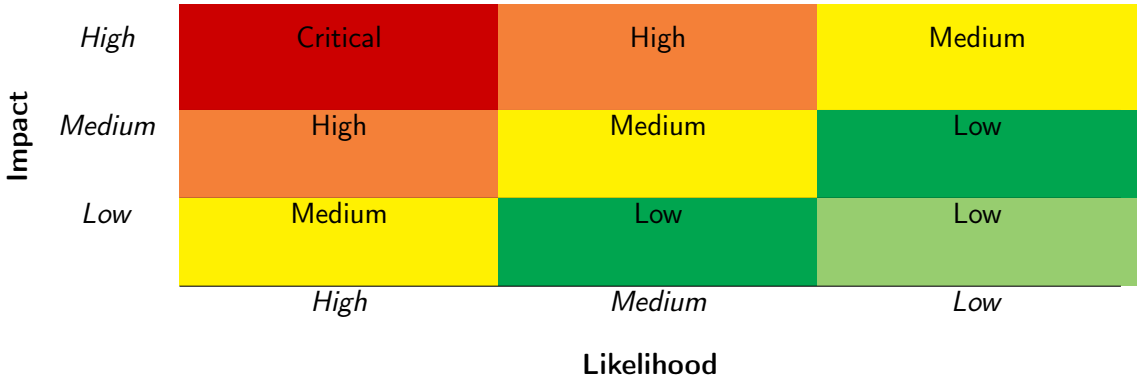
And here is the commit ID after fixes for the issues found in the audit have been checked in:

- https://github.com/revert-finance/v4utils.git (43f78fe, 82b9996)

1.2 About PeckShield

PeckShield Inc. [7] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [6]:

- 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) [5], 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 new `V4Utils` contract in `Revert`. 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	1	■
Low	2	■■
Informational	0	
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 1 medium-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key Revert V4Utils Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incompatible Encoding of SETTLE_PAIR Parameters	Coding Practices	Resolved
PVE-002	Low	Revisited Hook And Return Data Uses in V4Utils	Coding Practices	Resolved
PVE-003	Medium	Improper amountAddMin0 And amountAddMin1 Enforcement in V4Utils	Business Logic	Resolved

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 Incompatible Encoding of SETTLE_PAIR Parameters

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Swapper, V4Utils`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

By design, the new `V4Utils` contract interacts with the `UniswapV4 PositionManager (PosM)` to manage `LP` positions with user authorization. As part of the interaction, it supports one specific pool action, namely `SETTLE_PAIR`, to repay remaining unresolved amount (from current locker). Our analysis shows the prepared parameter for this pool action is incompatible from the expected input from `UniswapV4 PositionManager`.

In the following, we show below the implementation of the related `_buildActionsForIncreasingLiquidity()` routine. As the name indicates, this routine prepares the required parameter arrays with the given pool action sequence. The `SETTLE_PAIR` pool action is encoded right after the base action with its parameters encoded as `abi.encode(token0, token1, address(this))`. In other words, it currently encodes three arguments with `token0`, `token1`, and `address(this)`. Our analysis shows the third argument is not needed as the `SETTLE_PAIR` pool action only takes two argument of `token0` and `token1`.

```
185     function _buildActionsForIncreasingLiquidity(
186         uint8 baseAction,
187         Currency token0,
188         Currency token1
189     ) internal view returns (bytes memory actions, bytes[] memory params_array) {
190         if (token0.isAddressZero()  token1.isAddressZero()) {
191             // Include SWEEP action for native ETH
192             actions = abi.encodePacked(baseAction, uint8(Actions.SETTLE_PAIR), uint8(
                    Actions.SWEEP));
193             params_array = new bytes[](3);
194             params_array[2] = abi.encode(address(0), address(this));
```

```
195         } else {
196             // Standard actions for ERC20 tokens only
197             actions = abi.encodePacked(baseAction, uint8(Actions.SETTLE_PAIR));
198             params_array = new bytes[](2);
199         }
200         params_array[1] = abi.encode(token0, token1, address(this));
201     }
```

Listing 3.1: `Swapper::_buildActionsForIncreasingLiquidity()`

Moreover, inside the `_swapAndIncrease()` routine, while it invokes the above routine to prepare the required parameter sequences, it additionally overwrites the second pool action, i.e., `SETTLE_PAIR`, with `poolKey.currency0`, `poolKey.currency1`, and `address(this)`. This overwrite is redundant and can be removed.

Recommendation Improve the above routine by not encoding/overwriting the redundant argument.

Status This issue has been fixed by the following commit: `27fd270`.

3.2 Revisited Hook And Return Data Uses in V4Utils

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `V4Utils`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

The interaction with `UniswapV4` pools supports an extra `hook data` parameter to contain whatever context information that may be desired for interaction. This `hook data` parameter is officially supported in `UniswapV4 pools`. While reviewing the use of this `hook data` parameter, we notice it is being ignored in one specific instance, i.e., `_swapAndMint()`.

In the following, we show below the code snippet from this `_swapAndMint()` routine. By design, this routine is used to swap and mint a new LP position. Since a new LP `token id` will be minted (from the `MINT_POSITION` pool action), the associated action parameter contains a `hook data` parameter, which should be `params.mintHookData`, instead of current `bytes("")` (line 553).

```
524     function _swapAndMint(SwapAndMintParams memory params)
525         internal
526         returns (uint256 tokenId, uint128 liquidity, uint256 added0, uint256 added1)
527     {
528         (uint256 total0, uint256 total1) = _swapAndPrepareAmounts(params);
```

```
529
530         // V4 uses different approach - need to create PoolKey and use modifyLiquidities
531         PoolKey memory poolKey = PoolKey({
532             currency0: params.token0,
533             currency1: params.token1,
534             fee: params.fee,
535             tickSpacing: params.tickSpacing, // Use dynamic tickSpacing from params
536             hooks: IHooks(params.hook) // Use hook from params
537         });
538
539         (bytes memory actions, bytes[] memory params_array) =
540             _buildActionsForIncreasingLiquidity(uint8(Actions.MINT_POSITION), params.
                token0, params.token1);
541
542         // Calculate liquidity from amounts
543         liquidity = _calculateLiquidity(params.tickLower, params.tickUpper, poolKey,
                total0, total1);
544
545         params_array[0] = abi.encode(
546             poolKey,
547             params.tickLower,
548             params.tickUpper,
549             liquidity, // liquidity
550             total0, // amount0Max
551             total1, // amount1Max
552             address(this), // recipient
553             bytes("") // hookData
554         );
555         ...
556     }
```

Listing 3.2: `V4Utils::_swapAndMint()`

Recommendation Revisit the above routine to properly pass along the `hook data` parameter.

Status This issue has been fixed by the following commit: `27fd270`.

3.3 Improper amountAddMin0 And amountAddMin1 Enforcement in V4Utils

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `V4Utils`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

Among supported pool actions, `V4Utils` allows for seamless liquidity addition to chosen user positions. The liquidity addition is accompanied with necessary slippage control in given parameters, i.e., `amountAddMin0` and `amountAddMin1`. In the process of examining their enforcement, we notice an improvement in current implementation.

To elaborate, we use the `_swapAndIncrease()` function as an example and show below the related code snippet. Specifically, it computes the new liquidity amount (line 639) from the provided token assets `total0` and `total1`. The consumed tokens are then calculated as `added0 = total0 - finalBalance0` (line 654) and `added1 = total1 - finalBalance1` (line 655), where `finalBalance0/1` are current token balances of the `V4Utils` contract after liquidity addition. It comes to our attention that the consumed token assets should be computed as the difference before and after the liquidity addition. With that, we need to measure the token balances right before the liquidity addition and use them to compute the consumed token assets. Note the same issue also affects the `_swapAndMint()` function.

```
638         // Calculate liquidity from amounts
639         liquidity = _calculateLiquidity(info.tickLower(), info.tickUpper(), poolKey,
                total0, total1);
640
641         params_array[0] = abi.encode(params.tokenId, liquidity, total0, total1, params.
                increaseLiquidityHookData);
642         params_array[1] = abi.encode(poolKey.currency0, poolKey.currency1, address(this)
                );
643
644         positionManager.modifyLiquidities{value: address(this).balance}(
645             abi.encode(actions, params_array), params.deadline
646         );
647
648         // Calculate consumption and return leftovers
649         {
650             uint256 finalBalance0 = poolKey.currency0.balanceOfSelf();
651             uint256 finalBalance1 = poolKey.currency1.balanceOfSelf();
652
653             // Calculate amounts actually added
```

```
654             added0 = total0 - finalBalance0;
655             added1 = total1 - finalBalance1;
656
657             // Check minimum amounts were added
658             if (added0 < params.amountAddMin0) {
659                 revert InsufficientAmountAdded();
660             }
661             if (added1 < params.amountAddMin1) {
662                 revert InsufficientAmountAdded();
663             }
664             ...
665         }
```

Listing 3.3: `V4Utils::_swapAndIncrease()`

Recommendation Revisit the above-mentioned `_swapAndMint()` and `swapAndIncrease()` routines to properly enforce the slippage control when adding liquidity to user-chosen positions.

Status This issue has been fixed by the following commit: `43f78fe`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the new `V4Utils` contract in `Revert`, which provides streamlined tools to manage `UniswapV4 LP` positions. The `LP` positions may further serve as collateral and allow users to leverage by taking loans while retaining control and management of their capital within the `UniswapV4` pools. 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

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

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

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

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