SMART CONTRACT AUDIT REPORT

for

0xBund Router

Prepared By: Xiaomi Huang

PeckShield

November 7, 2025

Document Properties

Client	0xBund
Title	Smart Contract Audit Report
Target	0xBund Router
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 7, 2025	Xuxian Jiang	Final Release
1.0-rc	October 27, 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 source code of the 0xBund router smart contract, 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 0xBund Router

The 0xBund router handles order routing through 1inch and external DEX engines on both BNB Chain (BSC) and Ethereum mainnet. It's a single module and interacts with external swap routers and manages fee routing (typically 0.02%) and execution safety checks. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of 0xBund Router

Item	Description
Name	0xBund
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	November 7, 2025

In the following, we show the checksum values of the given package with smart contracts for audit.

- Router.sol: f8d2dcf03e14ddf7b10bf32a7380d3d9 (MD5)
- RouterAccessControl.sol: a10a3fcdfdf6056548c9128e7a0a02e5 (MD5)

- aesop-main-packages-evm.zip : 4af28246c918ce1dc74e2c5a4f84dbfb (MD5)

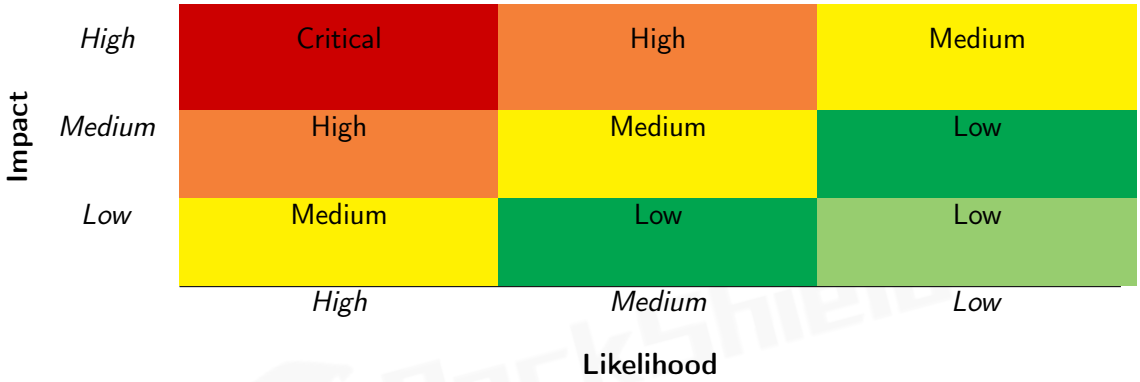
And here is the checksum hash value of the related repository after all fixes for the issues found in the audit have been applied:

- https://github.com/0xbund/aesop.git (9bf33d7)

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



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 accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

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

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) [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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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 implementation of the `0xBund router` smart contract. 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	2	■■
Low	2	■■
Informational	0	
Total	4	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of. 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 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, this smart contract is 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 2 low-severity vulnerabilities.

Table 2.1: Key 0xBund Router Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Inaccurate feeAmount Calculation in _swapExactOut()	Business Logic	Resolved
PVE-002	Low	Improved Token Allowance Management in Router	Coding Practice	Resolved
PVE-003	Low	input amount Unused Ether Return in _swapExactOut()	Coding Practice	Resolved
PVE-004	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 contract is being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Inaccurate feeAmount Calculation in _swapExactOut()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Router`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

While handling swap orders, `0xBund router` collects a certain swap fee, which will be calculated in the input token (if approved) or the output token (if the input token is not approved). While examining the fee amount calculation, we notice the handling of swap orders for exact output token amount has an issue in the fee amount calculation.

In the following, we show the code snippet from the related function - `_swapExactOut()`. As the name indicates, this function is used to execute an order for the expected output amount. When the input token is approved and the output token is not, the fee will be collected in the input token. As such, the fee amount should be computed as `totalAmountIn * routerFeeRate / (FEE_DENOMINATOR - routerFeeRate)`, not current `totalAmountIn * routerFeeRate / FEE_DENOMINATOR` (line 275).

```
273         if (isInputSupported && !isOutputSupported) {
274             // Collect fee from input amount
275             feeAmount = totalAmountIn * routerFeeRate / FEE_DENOMINATOR;
276             if (actualMaxAmountIn < totalAmountIn + feeAmount) revert Errors.
                    InsufficientInputAmount();
277
278             uint256 remainingInput = actualMaxAmountIn - totalAmountIn - feeAmount;
279
280             // Return unused tokens to user
281             if (remainingInput > 0) {
282                 if (msg.value > 0) {
283                     IWrappedToken(WRAPPED_TOKEN).withdraw(remainingInput);
284                     _transferNativeToken(params.to, remainingInput);
```

```
285                 } else {
286                     IERC20(inputToken).safeTransfer(params.to, remainingInput);
287                 }
288             }
289         }
```

Listing 3.1: `Router::_swapExactOut()`

Recommendation Revisit the above routine to compute the intended fee amount to collect.

Status This issue has been fixed in the following commit: `9bf33d7`.

3.2 Improved Token Allowance Management in Router

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Router`
- Category: Coding Practice [5]
- CWE subcategory: CWE-563 [2]

Description

The `0xBund router` contract is designed to interact with external `DEX` engines (e.g., `UniswapV2/V3`). In the process of reviewing the interaction, we notice the approved token allowance to the external `DEX` engines need to be reset especially when the given input token amounts are not used up.

In the following, we show the implementation of an example `_swapExactOutV2()` helper routine that is used to execute a swap on `UniswapV2` with exact output. If the given `amountInMax` of input token is not all used up, there is a need to invoke the following statement after the external swap, i.e., `IERC20(path[0]).safeApprove(v2Router, 0);`. Note another helper routine of `_swapExactOutV3()` shares the same issue.

```
372     function _swapExactOutV2(
373         uint256 amountInMax,
374         uint256 amountOut,
375         address[] calldata path,
376         address to,
377         uint256 deadline
378     ) internal returns (uint256) {
379         IUniswapV2Router02 v2RouterContract = IUniswapV2Router02(v2Router);
380         IERC20(path[0]).safeIncreaseAllowance(v2Router, amountInMax);
381         uint256[] memory amounts = v2RouterContract.swapTokensForExactTokens(
382             amountOut,
383             amountInMax,
384             path,
385             to,
```

```
386             deadline
387         );
388         return amounts[0];
389     }
```

Listing 3.2: `Router::_swapExactOutV2()`

Recommendation Revise the above-mentioned routines to reset token allowance after the token swap with external `DEX` engines.

Status This issue has been fixed in the following commit: `9bf33d7`.

3.3 Revisited Unused Ether Return in _swapExactOut()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Router`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, `0xBund router` allows to swap for the expected exact amount of output token. With that, there is a need to refund unused input tokens. Moreover, if the input token is native `Ether`, we better return the funds in native `Ether`. While analyzing the related refund logic, we notice a specific corner case where it returns unused native tokens in the wrapped form, not directly in `Ether`.

In the following, we show the code snippet from the related function - `_swapExactOut()`. When executing an order for the expected output amount, this function properly transfers input funds to the router contract. If the input token is the native `Ether`, the input funds are sent with the call together. With that, if there is any unused native `Ether`, the unused `Ether` is better returned in the form of `Ether`. While examining the below code snippet, we notice the unused input tokens are directly sent back via `IERC20(inputToken).safeTransfer(params.to, actualMaxAmountIn - totalAmountIn);` (line 293) in the form of wrapped input token, not native `Ether`.

```
291         if (isOutputSupported) {
292             // Transfer input amount to user
293             IERC20(inputToken).safeTransfer(params.to, actualMaxAmountIn - totalAmountIn
                    );
294             // Transfer output amount minus fee to user
295             uint256 remainingAmount = targetAmountOut - feeAmount;
296             // Check if output token is WETH and automatically convert to ETH
297             if (outputToken == WRAPPED_TOKEN) {
```

```
298                 IWrappedToken(WRAPPED_TOKEN).withdraw(remainingAmount);
299                 _transferNativeToken(params.to, remainingAmount);
300             } else {
301                 IERC20(outputToken).safeTransfer(params.to, remainingAmount);
302             }
303         }
```

Listing 3.3: `Router::_swapExactOut()`

Recommendation Revisit the above routine to properly return the used native `Ether`. An example revision is shown below.

```
291         if (isOutputSupported) {
292
293             uint256 remainingInput = actualMaxAmountIn - totalAmountIn;
294
295             // Return unused tokens to user
296             if (remainingInput > 0) {
297                 if (msg.value > 0) {
298                     IWrappedToken(WRAPPED_TOKEN).withdraw(remainingInput);
299                     _transferNativeToken(params.to, remainingInput);
300                 } else {
301                     IERC20(inputToken).safeTransfer(params.to, remainingInput);
302                 }
303             }
304
305             // Transfer output amount minus fee to user
306             uint256 remainingAmount = targetAmountOut - feeAmount;
307             // Check if output token is WETH and automatically convert to ETH
308             if (outputToken == WRAPPED_TOKEN) {
309                 IWrappedToken(WRAPPED_TOKEN).withdraw(remainingAmount);
310                 _transferNativeToken(params.to, remainingAmount);
311             } else {
312                 IERC20(outputToken).safeTransfer(params.to, remainingAmount);
313             }
314         }
```

Listing 3.4: Revised `Router::_swapExactOut()`

Status This issue has been fixed in the following commit: `9bf33d7`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Router`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

The `0xBund router` contract is designed with a privileged account, i.e., `admin`, that play a critical role in governing and regulating the router-wide operations (e.g., configure parameters, manage approved tokens, and withdraw funds from the router contract). It also has the privilege to control or govern the flow of assets managed by this router contract. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
608     function updateFeeRate(uint256 _newFeeRate) external onlyAdmin {
609         if (_newFeeRate > FEE_DENOMINATOR / 10) revert Errors.FeeRateTooHigh();
610         feeRate = _newFeeRate;
611         emit FeeRateUpdated(_newFeeRate);}
612     ...
613     function withdrawToken(
614         address token,
615         uint256 amount,
616         address recipient) public onlyAdmin {
617         if (IERC20(token).balanceOf(address(this)) < amount) revert Errors.
                InsufficientToken();
618         IERC20(token).safeTransfer(recipient, amount);}
619     ...
620     function approve(address router, address token, uint256 amount) external onlyAdmin {
621         _approve(router, token, amount);}
622     ...
623     function withdrawNativeToken(uint256 amount, address recipient) external onlyAdmin {
624         if (address(this).balance < amount) revert Errors.InsufficientToken();
625         _transferNativeToken(recipient, amount);}
626     ...
627     function addSupportedToken(address token) external onlyAdmin {
628         supportedTokens[token] = true;
629         emit TokenSupported(token);}
630     ...
631     function removeSupportedToken(address token) external onlyAdmin {
632         if (token == WRAPPED_TOKEN) revert Errors.InvalidInputToken();
633         supportedTokens[token] = false;
634         emit TokenUnsupported(token);
635     }
```

Listing 3.5: Example Privileged Operations in Router

We understand the need of the privileged function for contract maintenance, but at the same time the extra power to the privileged account may also be a counter-party risk to the protocol users. It is worrisome if the privileged account is plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

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 This issue has been mitigated with the use of a multi-sig for the privileged account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the 0xBund router contract that handles order routing through 1inch and external DEX engines on both BNB Chain (BSC) and Ethereum mainnet. It's a single module and interacts with external swap routers and manages fee routing (typically 0.02%) and execution safety checks. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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