# SMART CONTRACT AUDIT REPORT

for

# WidoRouter

Prepared By: Xiaomi Huang

PeckShield

December 21, 2022

## Document Properties

| | |
|---|---|
| Client | Wido |
| Title | Smart Contract Audit Report |
| Target | WidoRouter |
| Version | 1.0 |
| Author | Luck Hu |
| Auditors | Luck Hu, Xuxian Jiang |
| Reviewed by | Patrick Lou |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | December 21, 2022 | Luck Hu | Final Release |
| 1.0-rc | December 19, 2022 | Luck Hu | 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 `WidoRouter` 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 is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

## 1.1 About WidoRouter

`Wido` is a routing protocol that finds the best path to swap one token for another. It supports non-liquid tokens (e.g., vaults, pools, or farms) and enables single transaction deposits and withdrawals between vaults on `EVM` chains. The audited `WidoRouter` contract provides general interfaces for users to execute the orders for token transformations. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of WidoRouter

| Item | Description |
|---:|---|
| Name | Wido |
| Website | https://www.joinwido.com/ |
| Type | EVM Smart Contract |
| Platform | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | December 21, 2022 |

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the audit only covers `WidoRouter.sol` and `WidoTokenManager.sol`.

- https://github.com/widolabs/wido-contracts.git (e206636)

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

- https://github.com/widolabs/wido-contracts.git (007e3b8)

## 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 |

**Likelihood**

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on the 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 checklist of 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

Table 1.3: The Full Audit Checklist

| Category | Checklist Items |
|---|---|
| **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 |

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 Logic** | 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 `WidoRouter` 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

| Severity | # of Findings | |
|---|---|---|
| Critical | 0 | |
| High | 1 | ■ |
| Medium | 0 | |
| Low | 2 | ■ ■ |
| Informational | 0 | |
| Total | 3 | |

We have so far identified a list of potential issues. 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, these smart contract is well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key WidoRouter Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Low | Improved FulfilledOrder Event Generations | Coding Practices | Fixed |
| PVE-002 | High | Revisited Logic in receive() | Business Logic | Fixed |
| PVE-003 | Low | 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 Improved FulfilledOrder Event Generations

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `WidoRouter`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

### Description

In the `Wido` protocol, the `WidoRouter` contract provides the interfaces for users to execute their orders for token transformations. When one order is fulfilled, it emits a `FulfilledOrder` event. While reviewing the parameters in each of the emitted `FulfilledOrder` event, we notice the parameter orders of the second and the third parameters are inconsistent with the event definition.

To elaborate, we show below the code snippet of the `executeOrder()` routine, which is used to execute the given order. The `FulfilledOrder` event is emitted at the end of the routine (line 75), where the second and the third parameters are filled with `msg.sender` (transaction sender) and `order.user` (order recipient). However, in the `FulfilledOrder` event definition (lines 59 – 64), the second and the third parameters shall be the recipient and the transaction sender.

```
53      /// @notice Event emitted when the order is fulfilled
54      /// @param order The order that was fulfilled
55      /// @param recipient Recipient of the final tokens of the order
56      /// @param sender The msg.sender
57      /// @param feeBps Fee in basis points (bps)
58      /// @param partner Partner address
59      event FulfilledOrder(
60          Order order,
61          address recipient,
62          address indexed sender,
63          uint256 feeBps,
64          address indexed partner
65      );
```

```
    function executeOrder(
        Order calldata order,
        Step[] calldata route,
        uint256 feeBps,
        address partner
    ) external payable override nonReentrant {
        require(msg.sender == order.user, "Invalid order user");
        _executeOrder(order, route, order.user, feeBps);
        emit FulfilledOrder(order, msg.sender, order.user, feeBps, partner);
    }
```

Listing 3.1: WidoRouter::executeOrder()

Note the same issue is also applicable to the `executeOrderWithSignature()` routine.

**Recommendation** Reverse the second and the third parameters in the emitting of the `FulfilledOrder` event.

**Status** The issue has been fixed by this commit: `e77e312`.

## 3.2 Revisited Logic in receive()

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `WidoRouter`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

### Description

As mentioned in Section 3.1, the `WidoRouter` contract provides the interfaces for users to execute their orders for token transformations. It supports normal `ERC20` tokens as sell as the native token. Specifically, when the target token is the native token, the `WidoRouter` receives the native token from the target address and forwards the native token to the order recipient. While examining the logic to support the native token, we notice the `WidoRouter` contract can not receive the native token from the target address.

To elaborate, we show below the code snippet of the `receive()` routine, which is used to receive the native token. In the `receive()` routine, there is a validation for the `msg.sender` (line 333) and the `msg.sender` must be the `wrappedNativeToken` which is a wrapper for the native token. That is to say, the `WidoRouter` contract is implemented to receive the native token from the `wrappedNativeToken` only. As a result, the target address (e.g., `UniswapV2Router02`) can not send the native token to the `WidoRouter` contract, and the order execution reverts.

```
    /// @notice Reverts if the native tokens are sent directly to the contract
    receive() external payable {
        require(msg.sender == wrappedNativeToken);
    }
```

Listing 3.2: WidoRouter::receive()

**Recommendation** Revisit the `receive()` routine to support the receiving of the native token from the supported target addresses.

**Status** The issue has been fixed by this commit: `007e3b8`.

## 3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `WidoRouter`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

### Description

In the `WidoRouter` contract, there is a privileged account, i.e., `owner`, that plays a critical role in governing and regulating the system-wide operations. Our analysis shows that this privileged account needs to be scrutinized. In the following, we show the representative function potentially affected by the privileges of the `owner` account.

Specifically, the privileged function in `WidoRouter` allows for the `owner` to set the bank address, which is used to receive the protocol fees.

```
    /// @notice Sets the bank address
    /// @param _bank The address of the new bank
    function setBank(address _bank) external onlyOwner {
        require(_bank != address(0), "Bank address cannot be zero address");
        bank = _bank;
```

Listing 3.3: Example Privileged Operations in the `WidoRouter` Contract

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the `owner` may also be a counter-party risk to the protocol users. It is worrisome if the privileged `owner` account is a 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 changes to the 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 as the team confirmed that they plan to move the ownership to the multi-sig.

# 4 | Conclusion

In this audit, we have analyzed the design and implementation of the `WidoRouter` contract of `Wido`. `Wido` is a routing protocol that finds the best path to swap from one token to another. It supports non-liquid tokens (e.g., vaults, pools, or farms) and enables single transaction deposits and withdrawals between vaults on `EVM` chains. The audited `WidoRouter` contract provides general interfaces for users to execute the orders for token transformations. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.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.