



# SMART CONTRACT AUDIT REPORT

for

## XSwapRouter



Prepared By: Xiaomi Huang

PeckShield  
August 21, 2025

## Document Properties

Client	XSwapRouter
Title	Smart Contract Audit Report
Target	XSwapRouter
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	August 21, 2025	Xuxian Jiang	Final Release
1.0-rc1	August 21, 2025	Xuxian Jiang	Release Candidate #1

## Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Email	contact@peckshield.com

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	About XSwapRouter . . . . .	4
1.2	About PeckShield . . . . .	5
1.3	Methodology . . . . .	5
1.4	Disclaimer . . . . .	9
<b>2</b>	<b>Findings</b>	<b>10</b>
2.1	Summary . . . . .	10
2.2	Key Findings . . . . .	11
<b>3</b>	<b>Detailed Results</b>	<b>12</b>
3.1	Improved Pool Length Validation in V3Path . . . . .	12
3.2	Implicit Assumption Enforcement In addLiquidity() . . . . .	13
3.3	Revisited FLAG_ALLOW_REVERT Handling in XSwapRouter . . . . .	15
3.4	Inexhaustive Command Handling in dispatch() . . . . .	17
<b>4</b>	<b>Conclusion</b>	<b>18</b>
	<b>References</b>	<b>19</b>

# 1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the `XSwapRouter` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

## 1.1 About XSwapRouter

`XSwapRouter` is the core routing component of the `xSwap` protocol. It adopts a modular design and integrates multiple decentralized exchange functionalities to provide users with secure and efficient token swap services. Its modular design allows the same contract architecture to be deployed across different blockchains. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of XSwapRouter

Item	Description
Name	XSwapRouter
Website	<a href="https://xswap.app/">https://xswap.app/</a>
Type	Ethereum Smart Contracts
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 21, 2025

In the following, we show the deployment address of `XSwapRouter` used in this audit.

- XSwapRouter: <https://bscscan.com/address/0x9842A8eB5e42e63a87B3BBeb2D46B9B1f3fEE17C>

And this is the hash value of the final compressed file after all fixes for the issues found in the audit have been checked in:

- `contracts.zip`: MD5 (31a37dc23f6f0c30f65b5d15724f33b2)

## 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](mailto:contact@peckshield.com)).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on the 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 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 is considered safe regarding the check item. For any discovered issue, we might further deploy

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

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. 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.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
<b>Configuration</b>	Weaknesses in this category are typically introduced during the configuration of the software.
<b>Data Processing Issues</b>	Weaknesses in this category are typically found in functionality that processes data.
<b>Numeric Errors</b>	Weaknesses in this category are related to improper calculation or conversion of numbers.
<b>Security Features</b>	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
<b>Time and State</b>	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.
<b>Error Conditions, Return Values, Status Codes</b>	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.
<b>Resource Management</b>	Weaknesses in this category are related to improper management of system resources.
<b>Behavioral Issues</b>	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
<b>Business Logic</b>	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.
<b>Initialization and Cleanup</b>	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
<b>Arguments and Parameters</b>	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
<b>Expression Issues</b>	Weaknesses in this category are related to incorrectly written expressions within code.
<b>Coding Practices</b>	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.



## 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.



## 2 | Findings

### 2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `XSwapRouter` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	0	
Low	4	
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, 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 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 4 low-severity vulnerabilities.

Table 2.1: Key XSwapRouter Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Pool Length Validation in V3Path	Coding Practices	Confirmed
PVE-002	Low	Implicit Assumption Enforcement In addLiquidity()	Business Logic	Resolved
PVE-003	Low	Revisited FLAG_ALLOW_REVERT Handling in XSwapRouter	Coding Practices	Confirmed
PVE-004	Low	Inexhaustive Command Handling in dispatch()	Business Logic	Resolved

Besides 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 Pool Length Validation in V3Path

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: V3Path
- Category: Coding Practices [3]
- CWE subcategory: CWE-1126 [1]

#### Description

The XSwapRouter contract has a convenient library `V3Path`, which is developed to manipulate path data for multihop swaps. In the process of examining its logic to decode a swap path with possibly multiple `UniswapV3` pools, we notice current implementation may be improved.

To elaborate, we show below the code snippet of two related routines, i.e., `getFirstPool()` and `skipToken()`. As their names indicate, the former routine is used to get the segment corresponding to the first pool in the swap path and the latter is used to skip a `token + fee` element. With that, there is an implicit assumption in each routine. Specifically, the former requires `path.length >= Constants.V3_POP_OFFSET` and the latter assumes `path.length >= Constants.MULTIPLE_V3_POOLS_MIN_LENGTH`. These implicit assumptions are better resolved with respective explicit requirements.

```
34     function getFirstPool(  
35         bytes calldata path  
36     ) internal pure returns (bytes calldata) {  
37         return path[:Constants.V3_POP_OFFSET];  
38     }  
39  
40     ...  
41  
42     function skipToken(  
43         bytes calldata path  
44     ) internal pure returns (bytes calldata) {  
45         return path[Constants.NEXT_V3_POOL_OFFSET:];
```

46

}

Listing 3.1: V3Path::getFirstPool()/skipToken()

Moreover, we notice another routine named `UniV2Library::getAmountsOut()` can be improved by enforcing the following statement, i.e., `if (swapFee > 10_000) revert InvalidAmount();`.

**Recommendation** Revisit the above-mentioned routines to ensure the implicit assumptions, if any, are resolved.

**Status** The issue has been confirmed.

## 3.2 Implicit Assumption Enforcement In `addLiquidity()`

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: V2SwapRouter
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

### Description

The `XSwapRouter` protocol allows a user to not only swap tokens, but also manage liquidity in `UniswapV2` pairs. Specifically, the `addV2Liquidity()` routine (see the code snippet below) is provided to add `_amountADesired` amount of `tokenA/tokenAId` and `_amountBDesired` amount of `tokenB` into the pool as liquidity via the `Router::addLiquidity()` routine. To elaborate, we show below the related code snippet.

```

40     function addV2Liquidity(
41         address factory,
42         address tokenA,
43         address tokenB,
44         uint256 amountADesired,
45         uint256 amountBDesired,
46         uint256 amountAMin,
47         uint256 amountBMin
48     ) external {
49         (address pair, uint256 amountA, uint256 amountB) = UniV2Library
50             .addLiquidity(
51                 factory,
52                 tokenA,
53                 tokenB,
54                 amountADesired,
55                 amountBDesired,
56                 amountAMin,
57                 amountBMin

```

```

58         );
59         tokenA.safeTransferFrom(msg.sender, pair, amountA);
60         tokenB.safeTransferFrom(msg.sender, pair, amountB);
61         IUniV2Pool(pair).mint(msg.sender);
62     }

```

Listing 3.2: V2SwapRouter::addV2Liquidity()

```

67     function addLiquidity(
68         address factory,
69         address tokenA,
70         address tokenB,
71         uint256 amountADesired,
72         uint256 amountBDesired,
73         uint256 amountAMin,
74         uint256 amountBMin
75     ) internal returns (address pair, uint256 amountA, uint256 amountB) {
76         pair = IUniV2Factory(factory).getPair(tokenA, tokenB);
77         if (pair == address(0)) {
78             pair = IUniV2Factory(factory).createPair(tokenA, tokenB);
79         }

81         (uint256 reserve0, uint256 reserve1, ) = IUniV2Pool(pair).getReserves();
82         (uint256 reserveA, uint256 reserveB) = tokenA < tokenB
83             ? (reserve0, reserve1)
84             : (reserve1, reserve0);

86         if (reserveA == 0 && reserveB == 0) {
87             (amountA, amountB) = (amountADesired, amountBDesired);
88         } else {
89             uint256 amountB0ptimal = quote(amountADesired, reserveA, reserveB);
90             if (amountB0ptimal <= amountBDesired) {
91                 if (amountB0ptimal < amountBMin) revert InsufficientAmount();
92                 (amountA, amountB) = (amountADesired, amountB0ptimal);
93             } else {
94                 uint256 amountA0ptimal = quote(
95                     amountBDesired,
96                     reserveB,
97                     reserveA
98                 );
99                 assert(amountA0ptimal <= amountADesired);
100                 if (amountA0ptimal < amountAMin) revert InsufficientAmount();
101                 (amountA, amountB) = (amountA0ptimal, amountBDesired);
102             }
103         }
104     }

```

Listing 3.3: UniV2Library::addLiquidity()

It comes to our attention that the `UniV2Library` has implicit assumptions on the `_addLiquidity()` routine. The above routine takes a few arguments, including `_amountADesired/_amountBDesired` and `_amountAMin/_amountBMin`. The former `_amountADesired/_amountBDesired` determine the desired

amounts for adding liquidity to the pool and the latter `_amountAMin/_amountBMin` determine the minimum amount of used assets. There are two implicit conditions, i.e., `_amountADesired >= _amountAMin` and `_amountBDesired >= _amountBMin`. However, if these two conditions are not met, current logic will not trigger reverts because the code above performs asymmetric checks for these amounts. Hence, without stating these assumptions, slippage control for some trades on Router may not be checked and may not be taken into account at all in certain scenarios.

**Recommendation** Make the requirement of `_amountADesired >= _amountAMin` and `_amountBDesired >= _amountBMin` explicitly in the `addLiquidity()` function.

**Status** The issue has been fixed by adding the suggested requirement.

### 3.3 Revisited FLAG\_ALLOW\_REVERT Handling in XSwapRouter

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: XSwapRouter, Payments
- Category: Coding Practices [3]
- CWE subcategory: CWE-1126 [1]

#### Description

The XSwapRouter contract is inspired by Uniswap's Universal Router and enables users to execute multiple trading operations in a single transaction through encoded commands. The encoded command allows to specify `FLAG_ALLOW_REVERT`, a flag to indicate whether the command requires successful execution. Our analysis shows among all supported commands, only one may return `false` but all others will either return `true` or `revert`. If a command is reverted, the whole transaction is still reverted, failing to honor the `FLAG_ALLOW_REVERT` flag.

In the following, we show the implementation of the related `execute()` routine. The internal `dispatch()` handler may return `false` only when the subcommand is `Commands.BALANCE_CHECK_ERC20`, which attempts to check the token balance of a given address.

```

47     function execute(
48         bytes calldata commands,
49         bytes[] calldata inputs
50     ) public payable override isNotLocked {
51         bool success;
52         bytes memory output;
53         uint256 numCommands = commands.length;
54         if (inputs.length != numCommands) revert LengthMismatch();

```

```

56      // loop through all given commands, execute them and pass along outputs as
      defined
57      for (
58          uint256 commandIndex = 0;
59          commandIndex < numCommands;
60          commandIndex++
61      ) {
62          bytes1 command = commands[commandIndex];

64          bytes calldata input = inputs[commandIndex];

66          (success, output) = dispatch(command, input);

68          if (!success && successRequired(command)) {
69              revert ExecutionFailed({
70                  commandIndex: commandIndex,
71                  message: output
72              });
73          }
74      }
75  }

```

Listing 3.4: XSwapRouter::execute()

Moreover, there is a core `Payments::pay()` routine that is designed to pay an amount of ETH or ERC20 to a recipient. And the `value` parameter may take the pre-defined `ActionConstants.CONTRACT_BALANCE` value, which is currently only honored for transferring ERC20 tokens (line 60), not ETH.

```

56      function pay(address token, address recipient, uint256 value) internal {
57          if (token == Constants.ETH) {
58              recipient.safeTransferETH(value);
59          } else {
60              if (value == ActionConstants.CONTRACT_BALANCE) {
61                  value = IERC20(token).balanceOf(address(this));
62              }
63              token.safeTransfer(recipient, value);
64          }
65      }

```

Listing 3.5: Payments::pay()

**Recommendation** Revisit the above `FLAG_ALLOW_REVERT` flag or the internal dispatch handler so that it allows each command to fail, without reverting the whole transaction.

**Status** The issue has been confirmed.



### 3.4 Inexhaustive Command Handling in dispatch()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Dispatcher
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

#### Description

The XSwapRouter protocol has a core Dispatcher contract that performs the requested command execution. In the process of analyzing the list of supported commands, we notice the dispatch routine misses one specific command, i.e., 0x20.

In the following, we show the code snippet of the related dispatch() routine. In essence, the shown code snippet is used to examine the given command `commandType` (line 57) and execute the intended handling logic accordingly. Note the command examination is based on four `if` statements to cover possible commands. Our analysis shows it still misses a possible command with `commandType` = 0x20. To accommodate it, we can add one more `else` statement at the end with the following branch payload, i.e., `revert InvalidCommandType(command);`.

```

53     function dispatch(
54         bytes1 commandType,
55         bytes calldata inputs
56     ) internal returns (bool success, bytes memory output) {
57         uint256 command = uint8(commandType & Commands.COMMAND_TYPE_MASK);

59         success = true;

61         // First branch: 0x00 <= command < 0x08
62         if (command < 0x08) {...}
63         // Second branch: 0x08 <= command < 0x10
64         else if (command < 0x10) {...}
65         // Third branch: 0x10 <= command < 0x20
66         else if (command < 0x20) {...}
67         // Fourth branch: 0x21 <= command <= 0x3f (Advanced commands)
68         else if (command >= 0x21) {...}
69     }

```

Listing 3.6: Dispatcher::dispatch()

**Recommendation** Revisit the above routine to ensure all possible commands are handled.

**Status** The issue has been fixed by following the above suggestion.

## 4 | Conclusion

In this audit, we have analyzed the design and implementation of the `XSwapRouter` contract, which is the core routing component of the `XSwap` protocol. It adopts a modular design and integrates multiple decentralized exchange functionalities to provide users with secure and efficient token swap services. Its modular design allows the same contract architecture to be deployed across different blockchains. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.



## References

- [1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. <https://cwe.mitre.org/data/definitions/1126.html>.
- [2] MITRE. CWE-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](https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology).
- [7] PeckShield. PeckShield Inc. <https://www.peckshield.com>.