



SMART CONTRACT AUDIT REPORT

for

Swing Aggregator



Prepared By: Xiaomi Huang

PeckShield
August 11, 2024

Document Properties

Client	Swing
Title	Smart Contract Audit Report
Target	Swing Aggregator
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 11, 2024	Xuxian Jiang	Final Release
1.0-rc	July 9, 2024	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	4
1.1	About Swing Aggregator	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Possible Stealing of Funds From Approving Users	11
3.2	Possible Unsafe Calculation Logic in SwitchView	12
3.3	Improved Validation on Protocol Arguments	14
3.4	Trust Issue of Admin Keys	15
4	Conclusion	17
	References	18

1 | Introduction

Given the opportunity to review the design document and related source code of the `Swing` Aggregator 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 could potentially be improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Swing Aggregator

The `Swing` protocol provides the chain abstraction layer between L1s and L2s. It allows anyone to swap, stake, deposit digital assets on any protocol on any blockchain. It powers the smart routing which provides the best rates for users across chains, bridges, DEXes, and DeFi protocols. This audit covers the aggregator feature that powers the flexible cross-chain swaps and transfers. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Swing Aggregator

Item	Description
Name	Swing
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	August 11, 2024

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that the repository has a number of contracts and this audit excludes the following contracts: `SwitchMultichain.sol`, `SwitchConnexReceiver.sol`, `SwitchConnexSender.sol` under the `contracts/bridges/` directory.

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

- ## 1.2 About PeckShield

Table 1.2: Vulnerability Severity Classification

High Medium

Likelihood

1.3 Methodology

- 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.

5/18

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) [6], 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 design and implementation of the `Swing Aggregator` protocol smart contracts. 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	1	
High	0	
Medium	2	
Low	1	
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 may involve 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 1 critical-severity vulnerability, 2 medium-severity vulnerabilities, and 1 low-severity vulnerability.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Critical	Possible Stealing of Funds From Approving Users	Security Features	Resolved
PVE-002	Low	Possible Unsafe Calculation Logic in SwitchView	Coding Practices	Confirmed
PVE-003	Medium	Improved Validation on Protocol Arguments	Coding Practices	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 contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Possible Stealing of Funds From Approving Users

- ID: PVE-001
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

The Swing Aggregator protocol has a number of bridge contracts that are provided to seamlessly interact with various cross-chain solutions. In the process of examining these bridge contracts, we notice a need of validating and whitelisting the calling targets so that no user funds will be at risk.

```
186     function swapExternal(  
187         IERC20 srcToken,  
188         DataTypes.SplitSwapInfo memory splitSwapData  
189     ) external {  
190         require(  
191             msg.sender == address(this),  
192             "Msg.sender can be contract it self"  
193         );  
194  
195         if (splitSwapData.spender == address(0) && !srcToken.isETH()) {  
196             // Manually transfer instead of approve  
197             srcToken.universalTransfer(  
198                 splitSwapData.swapContract,  
199                 splitSwapData.amount  
200             );  
201         } else {  
202             srcToken.universalApprove(  
203                 splitSwapData.spender,  
204                 splitSwapData.amount  
205             );  
206         }  
207         (bool success, ) = splitSwapData.swapContract.call{
```

```

208         value: srcToken.isETH() ? splitSwapData.amount : 0
209     }(splitSwapData.swapData);
210
211     require(success, "External swap failed");
212 }

```

Listing 3.1: SwapRouter::swapExternal()

To elaborate, we show above an example SwapRouter contract and its swapExternal() function. As the name indicates, this function is used to call an example contract given as splitSwapData.swapContract and ensure the call is successful. However, the calling target is not validated, which may be exploited to drain funds from users who have approved funds to this SwapRouter contract. Note this issue affects a number of existing bridge contracts, including SwitchAcross, SwitchCelerSender/Receiver, and others.

Recommendation Improve the above logic to ensure the calling targets are whitelisted and properly validated.

Status The issue has been fixed by the following PRs: 226 and 243.

3.2 Possible Unsafe Calculation Logic in SwitchView

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: SwitchView
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [3]

Description

The Swing Aggregator protocol has a helper contract SwitchView that facilitates the calculation of return amount after various swap operations. While reviewing the return amount calculation, we notice current calculation may not be safe.

For example, if we examine closely the calculate() routine, it has a rather straightforward logic in computing the output token amount after the expected swap. We notice the calculation is delegated to an internal helper routine _calculate(), which relies on the token balances of the swap pair exchange (line 232). Note the pair's token balances may be easily impacted by flashloans or simple donation, which makes the output amount calculation unreliable.

```

179     function calculate(CalculateArgs memory args) public view returns(uint256[] memory
180         rets) {
181         return _calculate(
182             args.fromToken,

```

```

182         args.destToken,
183         args.factory,
184         _linearInterpolation(args.amount, args.parts)
185     );
186 }

```

Listing 3.2: SwitchView::calculate()

```

218     function _calculate(
219         IERC20 fromToken,
220         IERC20 destToken,
221         IUniswapFactory factory,
222         uint256[] memory amounts
223     )
224     internal
225     view
226     returns (uint256[] memory rets)
227     {
228         rets = new uint256[](amounts.length);
229
230         IERC20 fromTokenReal = fromToken.isETH() ? weth : fromToken;
231         IERC20 destTokenReal = destToken.isETH() ? weth : destToken;
232         IUniswapExchange exchange = factory.getPair(fromTokenReal, destTokenReal);
233         if (address(exchange) != address(0)) {
234             uint256 fromTokenBalance = fromTokenReal.universalBalanceOf(address(exchange));
235             uint256 destTokenBalance = destTokenReal.universalBalanceOf(address(exchange));
236             for (uint i = 0; i < amounts.length; i++) {
237                 rets[i] = _calculateUniswapFormula(fromTokenBalance, destTokenBalance,
238                                                     amounts[i]);
239             }
240             return rets;
241         }
242     }

```

Listing 3.3: SwitchView::_calculate()

Recommendation Revisit the above logic with necessary slippage enforcement to ensure the output amount calculation is reliable and safe.

Status This issue has been confirmed.

3.3 Improved Validation on Protocol Arguments

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The Swing Aggregator protocol is no exception. Specifically, if we examine the `SwapRouter` contract, it has defined a number of protocol-wide risk parameters, such as `pathCount` and `dexCount`. In the following, we show the corresponding routines that allow for their changes.

```

401     function setPathCount(uint256 _pathCount) external onlyOwner {
402         pathCount = _pathCount;
403         emit PathCountSet(_pathCount);
404     }
405
406     function setPathSplit(uint256 _pathSplit) external onlyOwner {
407         pathSplit = _pathSplit;
408         emit PathSplitSet(_pathSplit);
409     }
410
411     function setFactories(address[] memory _factories) external onlyOwner {
412         dexCount = _factories.length;
413         for (uint256 i = 0; i < _factories.length; i++) {
414             factories.push(_factories[i]);
415         }
416         emit FactoriesSet(_factories);
417     }

```

Listing 3.4: `SwapRouter::setFactories()`

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, the above `setFactories()` routine needs to update `dexCount` as `dexCount += _factories.length;`, not current `dexCount += _factories.length;` (line 412). Note the same issue also affects another routine, i.e., `SwitchRoot::setFactories()`.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

Status The issue has been fixed by the following PR: 225.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

The Swing Aggregator protocol has a privileged account, i.e., owner, that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure protocol-wide risk parameters, whitelist DEC engines, and execute privileged operations). It also has the privilege to control or govern the flow of assets among various protocol components. In the following, we examine the privileged account and related privileged accesses in current contracts.

```

88     function setReward(address _reward) external onlyOwner {
89         reward = _reward;
90         emit RewardSet(_reward);
91     }

93     function setFeeCollector(address _feeCollector) external onlyOwner {
94         feeCollector = _feeCollector;
95         emit FeeCollectorSet(_feeCollector);
96     }

98     function setMaxPartnerFeeRate(uint256 _maxPartnerFeeRate) external onlyOwner {
99         require(_maxPartnerFeeRate <= 5000, "too large");
100         maxPartnerFeeRate = _maxPartnerFeeRate;
101         emit MaxPartnerFeeRateSet(_maxPartnerFeeRate);
102     }

104     function setDefaultSwingCut(uint256 _defaultSwingCut) external onlyOwner {
105         defaultSwingCut = _defaultSwingCut;
106         emit DefaultSwingCutSet(_defaultSwingCut);
107     }

109     function setSwitchEvent(ISwitchEvent _switchEvent) external onlyOwner {
110         switchEvent = _switchEvent;
111         emit SwitchEventSet(_switchEvent);
112     }

114     function whitelistDEX(address _dex, bool _whitelisted) external onlyOwner {
115         _whitelistDEX(_dex, _whitelisted);

```

116

}

Listing 3.5: Example Privileged Operations in `SwitchV2`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the `owner` privilege to the intended DAO-like governance contract. And 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 multisig as the admin.



4 | Conclusion

In this audit, we have analyzed the design and implementation of the `aggregator` feature in `Swing` which provides the chain abstraction layer between `L1s` and `L2s`. It allows anyone to swap, stake, deposit digital assets on any protocol on any blockchain. It powers the smart routing which provides the best rates for users across chains, bridges, `DEXes`, and `DeFi` protocols. This audit covers the `aggregator` feature that powers the flexible cross-chain swaps and transfers. 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-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [3] MITRE. CWE-563: Assignment to Variable without Use. <https://cwe.mitre.org/data/definitions/563.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 VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [7] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [8] PeckShield. PeckShield Inc. <https://www.peckshield.com>.