SMART CONTRACT AUDIT REPORT

for

Swing Aggregator

Prepared By: Xiaomi Huang

PeckShield

May 31, 2023

Document Properties

Client	Swing.xyz
Title	Smart Contract Audit Report
Target	Swing Aggregator
Version	1.2
Author	Stephen Bie
Auditors	Stephen Bie, Xiaotao Wu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.2	May 31, 2023	Stephen Bie	Final Release
1.1	May 5, 2023	Xiaotao Wu	Final Release
1.0	March 27, 2023	Xiaotao Wu	Final Release

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 `Swing` 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 Swing

`Swing` is a decentralized DEX protocol that provides an aggregated access for single or cross-chains swap. It comes with great convenience in supporting a number of bridge protocols, including `Celer` , `Nxtp`, `MultiChain`, `Across`, `DeBridge`, `Hop`, `Hyphen`, and `Stargate`. The aggregation performs token swap aggregation across DEXes and supports crosschain token swap in a uniformed way. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Swing Aggregator

Item	Description
Name	Swing.xyz
Website	https://swing.xyz/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 31, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- https://github.com/polkaswitch/AggregatorContracts.git (8eb203e)

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

- https://github.com/polkaswitch/AggregatorContracts/commits/audit3 (cd017b5)

1.2 About PeckShield

PeckShield Inc. [11] 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 OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

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

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) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

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

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `Swing` 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 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	3	■ ■ ■
Informational	1	■
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 informational suggestion.

Table 2.1: Key Swing Aggregator Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Revisited Implementation Logic in SwitchNxtp::setTransactionManager()	Business Logic	Fixed
PVE-002	Medium	Lack of Slippage Control In Switch	Time and State	Fixed
PVE-003	Informational	Accommodation Of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-004	Low	Improved Sanity Checks in SwitchCelerSender::swapByCeler()	Coding Practices	Fixed
PVE-005	Low	Incorrect Implementation Logic in executeMessageWithTransfer()	Business Logic	Fixed
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Revisited Implementation Logic in SwitchNxtp::setTransactionManager()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SwitchNxtp`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

The `SwitchNxtp` contract provides an external `setTransactionManager()` function for the privileged `owner` account to change the `transactionManagerAddress`. While reviewing its logic, we notice the current implementation needs to be revisited.

In the following, we show the related code snippet of the `setTransactionManager()` routine. When the state variable `transactionManagerAddress` is changed, the state variable `transactionManager` should also be changed. Otherwise, the execution of the `transferByNxtp()/swapByNxtp()/swapByNxtpWithParaswap()` will revert.

```
66    function setTransactionManager(address _newTransactionManager) external onlyOwner {
67        transactionManagerAddress = _newTransactionManager;
68    }
```

Listing 3.1: `SwitchNxtp::setTransactionManager()`

Meanwhile, the `setTransactionManager()` function can also be improved to emit a related event, i.e., `SetTransactionManager(_newTransactionManager)` (right after line 67), when the `transactionManagerAddress/transactionManager` are being set.

Recommendation Properly revise the `setTransactionManager()` logic to add the `transactionManager` changing support. Meanwhile, emit the corresponding event.

Status This issue has been fixed in the following commit: `8ed20a`.

3.2 Lack of Slippage Control In Switch

- ID: PVE-002
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `Multiple contracts`
- Category: Time and State [6]
- CWE subcategory: CWE-362 [3]

Description

The `swapByNxtp()` function of the `SwitchNxtp` contract can support the swap from `srcSwap.srcToken` to `srcSwap.dstToken` before bridging via the `Nxtp`. While examining the implementation logic of this routine, we observe that there is no slippage control in place, which opens up the possibility for front-running and potentially results in a smaller `returnAmount` (line 131). Moreover, a similar issue also exists in the `SwitchStargateSender`, `SwitchStargateReceiver`, and `SwitchCelerReceiver` contracts.

```
108     function swapByNxtp(
109         SwapArgsNxtp calldata transferArgs,
110         bytes calldata encryptedCallData,
111         bytes calldata encodedBid,
112         bytes calldata bidSignature
113     )
114         external
115         payable
116         nonReentrant
117         returns (ITransactionManager.TransactionData memory)
118     {
119         require(transferArgs.recipient == msg.sender, "recipient must be equal to caller");
120         require(transferArgs.invariantData.receivingAddress == msg.sender, "recipient must be equal to caller");
121         require(transferArgs.expectedReturn >= transferArgs.minReturn, "expectedReturn must be equal or larger than minReturn");
122
123         IERC20(transferArgs.srcSwap.srcToken).universalTransferFrom(msg.sender, address(this), transferArgs.amount);
124         uint256 returnAmount = 0;
125         uint256 amountAfterFee = _getAmountAfterFee(IERC20(transferArgs.srcSwap.srcToken), transferArgs.amount, transferArgs.partner, transferArgs.partnerFeeRate);
126
127         // check fromToken is same or not destToken
128         if (transferArgs.srcSwap.srcToken == transferArgs.srcSwap.dstToken) {
129             returnAmount = amountAfterFee;
130         } else {
131             returnAmount = _swapBeforeNxtp(transferArgs, amountAfterFee);
132         }
133
134         if (returnAmount > 0) {
```

```
            uint256 approvedAmount = IERC20(transferArgs.srcSwap.dstToken).allowance(
                address(this), transactionManagerAddress);
            if (approvedAmount < returnAmount) {
                IERC20(transferArgs.srcSwap.dstToken).safeIncreaseAllowance(
                    transactionManagerAddress, returnAmount);
            }

            _emitCrossChainSwapRequest(transferArgs, returnAmount, msg.sender);

            return transactionManager.prepare(ITransactionManager.PrepareArgs({
                    invariantData: transferArgs.invariantData,
                    amount: returnAmount,
                    expiry: transferArgs.expiry,
                    encryptedCallData: encryptedCallData,
                    encodedBid: encodedBid,
                    bidSignature: bidSignature,
                    encodedMeta: "0x"
                })
            );
        } else {
            IERC20(transferArgs.srcSwap.srcToken).universalTransferFrom(address(this),
                msg.sender, transferArgs.amount);
            revert("Swap failed from dex");
        }
    }
```

Listing 3.2: SwitchNxtp::swapByNxtp()

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the liquidity provider. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above sandwich arbitrage to better protect the interests of users.

Status This issue has been fixed in the following commit: `8ed20a`.

3.3 Accommodation Of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
/**
* @dev Approve the passed address to spend the specified amount of tokens on behalf
    of msg.sender.
* @param _spender The address which will spend the funds.
* @param _value The amount of tokens to be spent.
*/
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

    // To change the approve amount you first have to reduce the addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it is not
    //  already 0 to mitigate the race condition described here:
    //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
}
```

Listing 3.3: `USDT Token Contract`

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

In the following, we use the `SwitchNxtp::transferByNxtp()` routine as an example. If the `USDT` token is supported as `transferArgs.fromToken`, the execution of `IERC20(transferArgs.fromToken).`

`safeIncreaseAllowance()` may revert if the old allowance and the new allowance to be configured are both non-zero (line 90).

```
70      function transferByNxtp(
71          TransferArgsNxtp calldata transferArgs,
72          bytes calldata encryptedCallData,
73          bytes calldata encodedBid,
74          bytes calldata bidSignature
75      )
76          external
77          payable
78          nonReentrant
79          returns (ITransactionManager.TransactionData memory)
80      {
81          require(transferArgs.invariantData.receivingAddress == msg.sender, "recipient
                must be equal to caller");

83          IERC20(transferArgs.fromToken).universalTransferFrom(msg.sender, address(this),
                transferArgs.amount);
84          uint256 amountAfterFee = _getAmountAfterFee(IERC20(transferArgs.fromToken),
                transferArgs.amount, transferArgs.partner, transferArgs.partnerFeeRate);

86          bool native = IERC20(transferArgs.fromToken).isETH();
87          if (!native) {
88              uint256 approvedAmount = IERC20(transferArgs.fromToken).allowance(address(
                    this), transactionManagerAddress);
89              if (approvedAmount < amountAfterFee) {
90                  IERC20(transferArgs.fromToken).safeIncreaseAllowance(
                        transactionManagerAddress, amountAfterFee);
91              }
92          }

94          _emitCrossChainTransferRequest(transferArgs, amountAfterFee, msg.sender);

96          return transactionManager.prepare(ITransactionManager.PrepareArgs({
97                  invariantData: transferArgs.invariantData,
98                  amount: amountAfterFee,
99                  expiry: transferArgs.expiry,
100                 encryptedCallData: encryptedCallData,
101                 encodedBid: encodedBid,
102                 bidSignature: bidSignature,
103                 encodedMeta: "0x"
104             })
105         );
106     }
```

Listing 3.4: `SwitchNxtp::transferByNxtp()`

Note the similar issue also exists in the `UniversalERC20::universalApprove()`, `Switch::_getAmountAfterFee()`, and `BaseTrade::_getAmountAfterFee()` routines.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`.

Status This issue has been fixed in the following commit: `8ed20a`.

3.4 Improved Sanity Checks in SwitchCelerSender::swapByCeler()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SwitchCelerSender`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

In the `SwitchCelerSender` contract, the `swapByCeler()` function is designed to facilitate the sending of a message to an app on another chain via `Celer MessageBus` (with an associated transfer). While reviewing the implementation of this routine, we notice that it can benefit from additional sanity checks.

To elaborate, we show below the code snippet of the `swapByCeler()` function. Specifically, there is a lack of sanity check for the input argument `transferArgs.srcSwap.dstToken`. If the given `transferArgs.srcSwap.dstToken` is the native token, the `MessageSenderLib.sendMessageWithTransfer()` execution will revert (lines 335-346).

```
279     function swapByCeler(
280         SwapArgsCeler calldata transferArgs
281     )
282         external
283         payable
284         nonReentrant
285         returns (bytes32 transferId)
286     {
287         require(transferArgs.recipient == msg.sender, "recipient must be equal to caller
                ");
288         require(transferArgs.expectedReturn >= transferArgs.minReturn, "expectedReturn
                must be equal or larger than minReturn");
289         IERC20(transferArgs.srcSwap.srcToken).universalTransferFrom(msg.sender, address(
                this), transferArgs.amount);
290 
291         uint256 returnAmount = 0;
292         uint256 amountAfterFee = _getAmountAfterFee(IERC20(transferArgs.srcSwap.srcToken
                ), transferArgs.amount, transferArgs.partner, transferArgs.partnerFeeRate);
293 
294         bytes memory message = abi.encode(
295             CelerSwapRequest({
```

```
                id: transferArgs.id,
                bridge: transferArgs.bridge,
                srcToken: transferArgs.srcSwap.srcToken,
                bridgeToken: transferArgs.dstSwap.srcToken,
                dstToken: transferArgs.dstSwap.dstToken,
                recipient: transferArgs.recipient,
                srcAmount: amountAfterFee,
                dstDistribution: transferArgs.dstDistribution,
                dstParaswapData: transferArgs.dstParaswapData,
                paraswapUsageStatus: transferArgs.paraswapUsageStatus,
                bridgeDstAmount: transferArgs.bridgeDstAmount,
                estimatedDstAmount: transferArgs.estimatedDstTokenAmount
            })
        );

        uint256 adjustedExecutionFee = getAdjustedExecutorFee(transferArgs.dstChainId);
        uint256 sgnFee = getSgnFeeByMessage(message);
        if (IERC20(transferArgs.srcSwap.srcToken).isETH()) {
            require(msg.value >= transferArgs.amount + sgnFee + adjustedExecutionFee, 'native token is not enough');
        } else {
            require(msg.value >= sgnFee + adjustedExecutionFee, 'native token is not enough');
        }

        payable(address(this)).transfer(adjustedExecutionFee);
        claimableExecutionFee += adjustedExecutionFee;

        if (transferArgs.srcSwap.srcToken == transferArgs.srcSwap.dstToken) {
            returnAmount = amountAfterFee;
        } else {
            if ((transferArgs.paraswapUsageStatus == DataTypes.ParaswapUsageStatus.OnSrcChain)  (transferArgs.paraswapUsageStatus == DataTypes.ParaswapUsageStatus.Both)) {
                returnAmount = _swapFromParaswap(transferArgs, amountAfterFee);
            } else {
                (returnAmount, ) = _swapBeforeCeler(transferArgs, amountAfterFee);
            }
        }

        require(returnAmount > 0, 'The amount too small');

        //MessageSenderLib is your swiss army knife of sending messages
        transferId = MessageSenderLib.sendMessageWithTransfer(
            transferArgs.callTo,
            transferArgs.srcSwap.dstToken,
            returnAmount,
            transferArgs.dstChainId,
            transferArgs.nonce,
            transferArgs.bridgeSlippage,
            message,
            MsgDataTypes.BridgeSendType.Liquidity,
```

```
            celerMessageBus,
            sgnFee
        );

        _emitCrossChainSwapRequest(transferArgs, transferId, returnAmount, msg.sender,
            DataTypes.SwapStatus.Succeeded);
    }
```

Listing 3.5: SwitchCelerSender::swapByCeler()

Note a similar issue also exists in the `depositByCeler()` routine of the same contract.

Recommendation Add necessary sanity checks to ensure the input argument `transferArgs.srcSwap.dstToken` is not native token for above mentioned functions.

Status This issue has been fixed in the following commit: `8ed20a`.

3.5 Incorrect Implementation Logic in executeMessageWithTransfer()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SwitchCelerDepositReceiver`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

The `SwitchCelerDepositReceiver` contract provides an `executeMessageWithTransfer()` handler which is required by the `Celer MessageBusReceiver`. While examining the `executeMessageWithTransfer()` routine of the `SwitchCelerDepositReceiver` contract, we notice the current implementation logic is not correct.

To elaborate, we show below the related code snippet. It comes to our attention that if the given `_token` is the native token, the transferred native token amount in line 114 is not correct. The correct native token amount to be transferred should be `depositAmount`, instead of current `_amount`.

```
86      function executeMessageWithTransfer(
87          address, //sender
88          address _token,
89          uint256 _amount,
90          uint64 _srcChainId,
91          bytes memory _message,
92          address // executor
93      )
94          external
95          payable
```

```
        onlyMessageBus
        returns (IMessageReceiverApp.ExecutionStatus)
    {
        CelerDepositRequest memory m = abi.decode((_message), (CelerDepositRequest));
        require(_token == m.bridgeToken, "bridged token must be the same as the first token in destination swap path");
        uint256 depositAmount = m.estimatedDepositAmount;
        if (m.bridgeToken != m.depositToken) {
            require(m.bridgeDstAmount <= _amount, "estimated bridge token balance is insufficient");
            // swap token through paraswap
            ICallDataExecutor(callDataExecutor).execute(IERC20(_token), augustusSwapper, paraswapProxy, _amount, 0, m.dstParaswapData);
            // deposit token to lending protocal
            ICallDataExecutor(callDataExecutor).execute(IERC20(m.depositToken), m.depositContract, m.toApprovalAddress, depositAmount, m.toContractGasLimit, m.depositCallData);
            _emitCrosschainDepositDone(m, _amount, depositAmount, DataTypes.DepositStatus.Succeeded);
        } else {
            depositAmount = m.bridgeDstAmount;
            require(depositAmount <= _amount, "deposit balance is insufficient");

            if (IERC20(_token).isETH()) {
                ICallDataExecutor(callDataExecutor).sendNativeAndExecute{ value: _amount }(
                    IERC20(m.depositToken),
                    m.depositContract,
                    m.toApprovalAddress,
                    depositAmount,
                    m.toContractGasLimit,
                    m.depositCallData
                );
            } else {
                // Give approval
                IERC20(_token).universalApprove(callDataExecutor, _amount);
                ICallDataExecutor(callDataExecutor).sendAndExecute(
                    IERC20(m.depositToken),
                    m.depositContract,
                    m.toApprovalAddress,
                    depositAmount,
                    m.toContractGasLimit,
                    m.depositCallData
                );
            }

            _sendToRecipient(_token, m.recipient, _amount - depositAmount);
            _emitCrosschainDepositDone(m, _amount, _amount, DataTypes.DepositStatus.Succeeded);
        }
        // always return true since swap failure is already handled in-place
        return IMessageReceiverApp.ExecutionStatus.Success;
```

```
    }
```

Listing 3.6: SwitchCelerDepositReceiver::executeMessageWithTransfer()

Recommendation Revisit the above mentioned function to correctly transfer the native token amount.

Status This issue has been fixed in the following commit: 8ed20a.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the Swing protocol, there is a privileged owner account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and owner adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we use the SwitchCelerSender contract as an example and show the representative functions potentially affected by the privileges of the owner account.

```
    function setCelerMessageBus(address _celerMessageBus) external onlyOwner {
        celerMessageBus = _celerMessageBus;
        emit CelerMessageBusSet(celerMessageBus);
    }

    function cBridgeSet(address _cBridge) external onlyOwner {
        cBridge = _cBridge;
        emit CelerMessageBusSet(cBridge);
    }

    function setPriceTracker(address _priceTracker) external onlyOwner {
        priceTracker = _priceTracker;
        emit PriceTrackerSet(priceTracker);
    }

    function setExecutorFee(uint256 _executorFee) external onlyOwner {
        require(_executorFee > 0, "price cannot be 0");
        executorFee = _executorFee;
        emit ExecutorFeeSet(_executorFee);
    }
```

```
    function claimExecutorFee(address feeReceiver) external onlyOwner {
        payable(feeReceiver).transfer(claimableExecutionFee);
        emit ExecutorFeeClaimed(claimableExecutionFee, feeReceiver);
        claimableExecutionFee = 0;
    }
```

Listing 3.7: Example Privileged Operations in `SwitchCelerSender`

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 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 confirms multi-sig address will be used.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Swing` protocol, which is a decentralized DEX protocol that provides an aggregated access for single or cross-chains swap. It comes with great convenience in supporting a number of bridge protocols, including `Celer`, `Nxtp`, `MultiChain`, `Across`, `DeBridge`, `Hop`, `Hyphen`, and `Stargate`. The aggregation performs token swap aggregation across DEXes and supports crosschain token swap in a uniformed way. 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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

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

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

[6] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

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

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

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