SMART CONTRACT AUDIT REPORT

for

FundMovr

Prepared By: Yiqun Chen

PeckShield
September 26, 2021

Document Properties

Client	Movr Network
Title	Smart Contract Audit Report
Target	FundMovr
Version	1.0-rc
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	September 26, 2021	Xiaotao Wu	Release Candidate #1

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the `FundMovr` 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 FundMovr

`FundMovr` is a bridge aggregator that allows for flexible cross-chain transfers. The goal of the `FundMovr` smart contracts is to execute the route provided by off-chain quoting engines. External integrations can either be `middlewares` or `bridges` and they need to follow the `MiddlewareImplBase` and `ImplBase` interfaces respectively. Implementations are linked in the `routes` array that can be updated by an external maintainer. Each `middleware` or `bridge` route is assigned an index in the array and is uniquely identified by the index.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of FundMovr

Item	Description
Name	Movr Network
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 26, 2021

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

- https://github.com/movrnetwork/aggregator-contracts (1d22955)

1.2 About PeckShield

PeckShield Inc. [8] 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 [7]:

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

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

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

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

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 `FundMovr` 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	0	
High	0	
Medium	2	■ ■
Low	2	■ ■
Informational	1	■
Total	5	

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 2 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improved Handling Of Corner Cases	Business Logic	Confirmed
PVE-002	Low	Incompatibility with Deflationary/Rebasing Tokens	Business Logic	Confirmed
PVE-003	Low	Potential Reentrancy Risks in Registry	Time and State	Confirmed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-005	Informational	Proper Allowance Cancellation	Business Logic	Confirmed

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 Handling Of Corner Cases

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: Low
- Target: `Multiple contracts`
- Category: Business Logic [5]
- CWE subcategory: CWE-837 [2]

Description

The `Registry` contract of the `FundMovr` protocol provides an external `outboundTransferTo()` function for users to call the respective implementation (depending on the bridge to be used). While examining its logic, we notice the current implementation can be improved.

To elaborate, we show below its code snippet. If the `_input.middlewareID` is 0, there is no swap required and we can directly bridge the source token (line 88). In this case, the `require` statement in lines 89-94 becomes unnecessary and the execution may revert.

```
73      /**
74      // @notice function responsible for calling the respective implementation
75      // depending on the bridge to be used
76      // If the middlewareId is 0 then no swap is required,
77      // we can directly bridge the source token to wherever required,
78      // else, we first call the Swap Impl Base for swapping to the required
79      // token and then start the bridging
80      // @dev It is required for isMiddleWare to be true for route 0 as it is a special
            case
81      // @param _input calldata follows the input data struct
82      */
83      function outboundTransferTo(InputData calldata _input) external payable {
84          require(_input.amount != 0, MovrErrors.INVALID_AMT);
85          require(_input.bridgeID != 0, MovrErrors.INVALID_BRIDGE_ID);
86
87          // read middleware info and validate
88          RouteData memory middlewareInfo = routes[_input.middlewareID];
89          require(
```

```
            middlewareInfo.route != address(0) &&
                middlewareInfo.enabled &&
                middlewareInfo.isMiddleware,
            MovrErrors.ROUTE_NOT_ALLOWED
        );

        // read bridge info and validate
        RouteData memory bridgeInfo = routes[_input.bridgeID];
        require(
            bridgeInfo.route != address(0) &&
                bridgeInfo.enabled &&
                !bridgeInfo.isMiddleware,
            MovrErrors.ROUTE_NOT_ALLOWED
        );

        // if middlewareID is 0 it means we dont want to perform a action before
            bridging
        // and directly want to move for bridging
        if (_input.middlewareID == 0) {
            // perform the bridging
            ImplBase(bridgeInfo.route).outboundTransferTo(
                _input.amount,
                msg.sender,
                _input.to,
                _input.tokenToBridge,
                _input.toChainId,
                _input.bridgeData
            );
        } else {
            // we perform an action using a middleware
            uint256 _amountOut =
                MiddlewareImplBase(middlewareInfo.route).performAction{
                    value: msg.value
                }(
                    msg.sender,
                    _input.middlewareInputToken,
                    _input.amount,
                    address(this),
                    _input.middlewareData
                );

            // we give allowance to the bridge
            IERC20(_input.tokenToBridge).safeIncreaseAllowance(
                bridgeInfo.route,
                _amountOut
            );

            // perform the bridging
            ImplBase(bridgeInfo.route).outboundTransferTo(
                _amountOut,
                address(this),
                _input.to,
```

```
                _input.tokenToBridge,
                _input.toChainId,
                _input.bridgeData
            );
        }

        emit ExecutionCompleted(_input.middlewareID, _input.bridgeID);
    }
```

Listing 3.1: `Registry::outboundTransferTo()`

Another corner case handling issue can be found in the `performAction()` routine of the `OneInchSwapImpl` contract. Specifically, if the `fromToken` is `address(0)`, the executions of `IERC20(fromToken).safeTransferFrom(from, address(this), amount)` and `IERC20(fromToken).safeIncreaseAllowance(oneInchAggregator, amount)` will revert (lines 52-53).

```
36      /**
37      // @notice Function responsible for swapping from one token to a different token
38      // @dev This is called only when there is a request for a swap.
39      // @param from userAddress or sending address.
40      // @param fromToken token to be swapped
41      // @param amount amount to be swapped
42      // param to not required. This is there only to follow the MiddlewareImplBase
43      // @param swapExtraData data required for the one inch aggregator to get the swap
            done
44      */
45      function performAction(
46          address from,
47          address fromToken,
48          uint256 amount,
49          address, // to
50          bytes memory swapExtraData
51      ) external payable override onlyRegistry returns (uint256) {
52          IERC20(fromToken).safeTransferFrom(from, address(this), amount);
53          IERC20(fromToken).safeIncreaseAllowance(oneInchAggregator, amount);
54          {
55              // solhint-disable-next-line
56              (bool success, bytes memory result) =
57                  oneInchAggregator.call{value: msg.value}(swapExtraData);
58
59              require(success, MovrErrors.MIDDLEWARE_ACTION_FAILED);
60              (uint256 returnAmount, ) = abi.decode(result, (uint256, uint256));
61              return returnAmount;
62          }
63      }
```

Listing 3.2: `OneInchSwapImpl::performAction()`

Recommendation Take into consideration the scenario where the value of `_input.middlewareID` might be equal to 0. Also, handle the corner case when the value of the `fromToken` might be equal to `address(0)`.

Status The issue has been confirmed.

3.2 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-2
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In `FundMovr`, the `OneInchSwapImpl` contract is designed to be called by the `Registry` before cross chain transfers if the user requests for a swap. In particular, one entry routine, i.e., `performAction()`, allows a user to transfer the specified assets (e.g., `fromToken`) to the `OneInchSwapImpl` contract and further swaps the `fromToken` to a different token for the user through `oneInchAggregator`. Naturally, the contract implements a number of low-level helper routines to transfer assets in the `OneInchSwapImpl` contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract. In the following, we show the `performAction()` routine that is used to transfer `IERC20(fromToken)` to the `OneInchSwapImpl` contract.

```
    /**
    // @notice Function responsible for swapping from one token to a different token
    // @dev This is called only when there is a request for a swap.
    // @param from userAddress or sending address.
    // @param fromToken token to be swapped
    // @param amount amount to be swapped
    // param to not required. This is there only to follow the MiddlewareImplBase
    // @param swapExtraData data required for the one inch aggregator to get the swap
        done
    */
    function performAction(
        address from,
        address fromToken,
        uint256 amount,
        address, // to
        bytes memory swapExtraData
    ) external payable override onlyRegistry returns (uint256) {
        IERC20(fromToken).safeTransferFrom(from, address(this), amount);
        IERC20(fromToken).safeIncreaseAllowance(oneInchAggregator, amount);
        {
            // solhint-disable-next-line
            (bool success, bytes memory result) =
                oneInchAggregator.call{value: msg.value}(swapExtraData);
```

```
                require(success, MovrErrors.MIDDLEWARE_ACTION_FAILED);
                (uint256 returnAmount, ) = abi.decode(result, (uint256, uint256));
                return returnAmount;
            }
        }
```

Listing 3.3: `OneInchSwapImpl::performAction()`

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `performAction()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the contract before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into `FundMovr` for trading. Meanwhile, there exist certain assets that may exhibit control switches that can be dynamically exercised to convert into deflationary. Note this issue is also present in the `outboundTransferTo()` routine of all `L1` and `L2` implementation contracts.

Recommendation If current codebase needs to support deflationary tokens, it is necessary to check the balance before and after the `transfer()`/`transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted `USDT`.

Status The issue has been confirmed.

3.3 Potential Reentrancy Risks in Registry

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Registry`
- Category: Time and State [5]
- CWE subcategory: CWE-841 [3]

Description

While reviewing the current `FundMovr` contracts, we notice there is a potential reentrancy risk in the `outboundTransferTo()` function of the `Registry` contract. To elaborate, we show below the code snippet of this routine. The execution logic is rather straightforward: if the middlewareId is 0, it directly bridges the source token to wherever required. Otherwise, it calls the `MiddlewareImplBase` contract for swapping to the required token and then starts the bridging.

```
    /**
    // @notice function responsible for calling the respective implementation
    // depending on the bridge to be used
    // If the middlewareId is 0 then no swap is required,
    // we can directly bridge the source token to wherever required,
    // else, we first call the Swap Impl Base for swapping to the required
    // token and then start the bridging
    // @dev It is required for isMiddleWare to be true for route 0 as it is a special case
    // @param _input calldata follows the input data struct
    */
    function outboundTransferTo(InputData calldata _input) external payable {
        require(_input.amount != 0, MovrErrors.INVALID_AMT);
        require(_input.bridgeID != 0, MovrErrors.INVALID_BRIDGE_ID);

        // read middleware info and validate
        RouteData memory middlewareInfo = routes[_input.middlewareID];
        require(
            middlewareInfo.route != address(0) &&
                middlewareInfo.enabled &&
                middlewareInfo.isMiddleware,
            MovrErrors.ROUTE_NOT_ALLOWED
        );

        // read bridge info and validate
        RouteData memory bridgeInfo = routes[_input.bridgeID];
        require(
            bridgeInfo.route != address(0) &&
                bridgeInfo.enabled &&
                !bridgeInfo.isMiddleware,
            MovrErrors.ROUTE_NOT_ALLOWED
        );
```

```

        // if middlewareID is 0 it means we dont want to perform a action before bridging
        // and directly want to move for bridging
        if (_input.middlewareID == 0) {
            // perform the bridging
            ImplBase(bridgeInfo.route).outboundTransferTo(
                _input.amount,
                msg.sender,
                _input.to,
                _input.tokenToBridge,
                _input.toChainId,
                _input.bridgeData
            );
        } else {
            // we perform an action using a middleware
            uint256 _amountOut =
                MiddlewareImplBase(middlewareInfo.route).performAction{
                    value: msg.value
                }(
                    msg.sender,
                    _input.middlewareInputToken,
                    _input.amount,
                    address(this),
                    _input.middlewareData
                );

            // we give allowance to the bridge
            IERC20(_input.tokenToBridge).safeIncreaseAllowance(
                bridgeInfo.route,
                _amountOut
            );

            // perform the bridging
            ImplBase(bridgeInfo.route).outboundTransferTo(
                _amountOut,
                address(this),
                _input.to,
                _input.tokenToBridge,
                _input.toChainId,
                _input.bridgeData
            );
        }

        emit ExecutionCompleted(_input.middlewareID, _input.bridgeID);
    }
```

Listing 3.4: `Registry::outboundTransferTo()`

However, our analysis shows that the current implementation of `outboundTransferTo()` lacks reentrancy prevention. If the underlying token faithfully implements the ERC777-like standard, then the `outboundTransferTo()` routine is vulnerable to reentrancy and this risk needs to be properly mitigated.

Specifically, the ERC777 standard normalizes the ways to interact with a token contract while remaining backward compatible with ERC20. Among various features, it supports send/receive hooks to offer token holders more control over their tokens. Specifically, when `transfer()` or `transferFrom()` actions happen, the owner can be notified to make a judgment call so that she can control (or even reject) which token they send or receive by correspondingly registering `tokensToSend` and `tokensReceived` hooks. Consequently, any `transfer()` or `transferFrom()` of ERC777-based tokens might introduce the chance for reentrancy or hook execution for unintended purposes (e.g., mining GasTokens).

In the ERC777 token case, the above hook can be planted in `ImplBase(bridgeInfo.route).outboundTransferTo()` and `MiddlewareImplBase(middlewareInfo.route).performAction()` (lines 109, 120 and 137) before the actual transfer of the underlying token occurs. In this particular case, if the external contract has certain hidden logic, we may run into risk of having a `re-entrancy` via other public methods.

Recommendation Add necessary reentrancy guards (e.g., `nonReentrant`) to prevent unwanted reentrancy risks.

Status The issue has been confirmed.

3.4 Trust Issue of Admin Keys

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

Description

In the `FundMovr` protocol, there is a certain privileged account, i.e., `_owner`. When examining the related contracts, we notice inherent trust on this privileged account. To elaborate, we show below the related functions.

Firstly, the `addRoutes()/updateRoute()` functions allow for the `_owner` to add/update routes to the `Registry`.

```
    /// @notice add routes to the registry.
    function addRoutes(RouteData[] calldata _routes)
        external
        onlyOwner
        returns (uint256[] memory)
    {
```

```
        require(_routes.length != 0, MovrErrors.EMPTY_INPUT);
        uint256[] memory _routeIds = new uint256[](_routes.length);
        for (uint256 i = 0; i < _routes.length; i++) {
            require(
                _routes[i].route != address(0),
                MovrErrors.ADDRESS_0_PROVIDED
            );
            routes.push(_routes[i]);
            _routeIds[i] = routes.length - 1;
            emit NewRouteAdded(
                i,
                _routes[i].route,
                _routes[i].enabled,
                _routes[i].isMiddleware
            );
        }

        return _routeIds;
    }

    ///@notice Updates the route data if required.
    function updateRoute(uint256 _routeId, RouteData calldata _routeData)
        external
        onlyOwner
        onlyExistingRoute(_routeId)
    {
        routes[_routeId] = _routeData;
        emit RouteUpdated(
            _routeId,
            _routeData.route,
            _routeData.enabled,
            _routeData.isMiddleware
        );
    }
```

Listing 3.5: `Registry::addRoutes()/updateRoute()`

Secondly, the `rescueFunds()` function allows for the `_owner` to transfer the `IERC20(token)` from the contracts to a specified user.

```
    function rescueFunds(
        address token,
        address userAddress,
        uint256 amount
    ) external onlyOwner {
        IERC20(token).safeTransfer(userAddress, amount);
    }
```

Listing 3.6: `Registry::rescueFunds()`

```
    function rescueFunds(
        address token,
        address userAddress,
```

```
        uint256 amount
    ) external onlyOwner {
        IERC20(token).safeTransfer(userAddress, amount);
    }
```

Listing 3.7: `MiddlewareImplBase::rescueFunds()`

```
    function rescueFunds(
        address token,
        address userAddress,
        uint256 amount
    ) external onlyOwner {
        IERC20(token).safeTransfer(userAddress, amount);
    }
```

Listing 3.8: `ImplBase::rescueFunds()`

Thirdly, the `updateRegistryAddress()` function allows for the `_owner` to update the `registry` address for the `L1` and `L2` implementation contracts.

```
    function updateRegistryAddress(address newRegistry) external onlyOwner {
        registry = newRegistry;
    }
```

Listing 3.9: `ImplBase::updateRegistryAddress()`

Fourthly, the `setRouter()/setrootChainManagerProxy()/setErc20PredicateProxy()` functions allow for the `_owner` to set `router` for the `NativeArbitrumImpl` contract and to set `rootChainManagerProxy` and `erc20PredicateProxy` for the `NativePolygonImpl` contract.

```
    /// @notice setter function for the L1 gateway router address
    function setRouter(address _router) public onlyOwner {
        router = _router;
    }
```

Listing 3.10: `NativeArbitrumImpl::setRouter()`

```
    /**
    // @notice Function to set the root chain manager proxy address.
     */
    function setrootChainManagerProxy(address _rootChainManagerProxy)
        public
        onlyOwner
    {
        rootChainManagerProxy = _rootChainManagerProxy;
    }

    /**
    // @notice Function to set the ERC20 Predicate proxy address.
     */
    function setErc20PredicateProxy(address _erc20PredicateProxy)
        public
```

```
        onlyOwner
    {
        erc20PredicateProxy = _erc20PredicateProxy;
    }
```

Listing 3.11: `NativePolygonImpl::setrootChainManagerProxy()/setErc20PredicateProxy()`

Lastly, the `setOneInchAggregator()` function allows for the `_owner` to set `oneInchAggregator` for the `OneInchSwapImpl` contract.

```
    /// @param _oneInchAggregator is the address for oneInchAggreagtor
    function setOneInchAggregator(address _oneInchAggregator)
        external
        onlyOwner
    {
        oneInchAggregator = payable(_oneInchAggregator);
    }
```

Listing 3.12: `OneInchSwapImpl::setOneInchAggregator()`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the `_owner` 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 Make the list of extra privileges granted to `_owner` explicit to `FundMovr` users.

Status The issue has been confirmed.

3.5 Proper Allowance Cancellation

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple contracts`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

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 3.13: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use the `OneInchSwapImpl::performAction()` routine as an example. This routine will increase a specific amount of `fromToken` token for the `oneInchAggregator` contract (line 53). 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. To accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()` and in case the allowance for the `oneInchAggregator` is not fully consumed by the swap (line 57), the allowance for the `oneInchAggregator` should be canceled after the swap execution.

```
36    /**
37    // @notice Function responsible for swapping from one token to a different token
38    // @dev This is called only when there is a request for a swap.
39    // @param from userAddress or sending address.
40    // @param fromToken token to be swapped
41    // @param amount amount to be swapped
42    // param to not required. This is there only to follow the MiddlewareImplBase
43    // @param swapExtraData data required for the one inch aggregator to get the swap
          done
44    */
45    function performAction(
46        address from,
47        address fromToken,
48        uint256 amount,
49        address, // to
50        bytes memory swapExtraData
51    ) external payable override onlyRegistry returns (uint256) {
```

```
        IERC20(fromToken).safeTransferFrom(from, address(this), amount);
        IERC20(fromToken).safeIncreaseAllowance(oneInchAggregator, amount);
        {
            // solhint-disable-next-line
            (bool success, bytes memory result) =
                oneInchAggregator.call{value: msg.value}(swapExtraData);

            require(success, MovrErrors.MIDDLEWARE_ACTION_FAILED);
            (uint256 returnAmount, ) = abi.decode(result, (uint256, uint256));
            return returnAmount;
        }
    }
```

Listing 3.14: `OneInchSwapImpl::performAction()`

The same issue also exists in the `outboundTransferTo()` routine of the `L1` and `L2` implementation contracts.

Recommendation Properly cancel the allowance for the `oneInchAggregator` after the swap is executed. An example revision is shown below.

```
    /**
    // @notice Function responsible for swapping from one token to a different token
    // @dev This is called only when there is a request for a swap.
    // @param from userAddress or sending address.
    // @param fromToken token to be swapped
    // @param amount amount to be swapped
    // param to not required. This is there only to follow the MiddlewareImplBase
    // @param swapExtraData data required for the one inch aggregator to get the swap
        done
    */
    function performAction(
        address from,
        address fromToken,
        uint256 amount,
        address, // to
        bytes memory swapExtraData
    ) external payable override onlyRegistry returns (uint256) {
        IERC20(fromToken).safeTransferFrom(from, address(this), amount);
        IERC20(fromToken).safeIncreaseAllowance(oneInchAggregator, amount);
        {
            // solhint-disable-next-line
            (bool success, bytes memory result) =
                oneInchAggregator.call{value: msg.value}(swapExtraData);

            IERC20(fromToken).safeApprove(oneInchAggregator, 0);
            require(success, MovrErrors.MIDDLEWARE_ACTION_FAILED);
            (uint256 returnAmount, ) = abi.decode(result, (uint256, uint256));
            return returnAmount;
        }
    }
```

Listing 3.15: `OneInchSwapImpl::performAction()`

Status This issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the `FundMovr` design and implementation. `FundMovr` is a bridge aggregator and allows for any-to-any cross-chain 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.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: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.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.