Security Audit Report for Multichain Smart Contracts

Date: August 12, 2022

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2022.08.13 00:44:37 +08'00'

Contents

Report Manifest

Item	Description
Client	Multichain
Target	Multichain Smart Contracts

Version History

Version	Date	Description
1.0	August 12, 2022	First Release

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 5 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the Multichain Smart Contracts for EVM-compatible chains [1]. The audit scope is limited to contracts under the following two folders: `contracts/access` and `contracts/router`. Besides, it is worth noting that Multichain Smart Contracts relies a decentralized network named *MPC* to forward messages among different blockchains. As confirmed by Multichain, MPC is trustworthy and out of the scope of this audit.

The auditing process is iterative. Specifically, we audit the initial version and following commits that fix the discovered issues. If there are new issues, we will continue this process. The commit hash values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Multichain Smart Contracts	`Version 1`	292dba33244de2204c693b32da786d3793b06bc4
	`Version 2`	6f9630d768a55a606572517be3982aabd395eec2
	`Version 3`	908ea136c87b1f2df4300911a703ad89ca8a087b

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

[1] https://github.com/anyswap/multichain-smart-contracts

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Access control
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find **three** potential issues. We have **two** recommendations.

- High Risk: 0
- Medium Risk: 3
- Low Risk: 0
- Recommendations: 2
- Notes: 1

ID	Severity	Description	Category	Status
1	Medium	Potential reentrancy in balance calculation	Software Security	Fixed
2	Medium	Incorrect logic in the `retrySwapinAndExec` function	Software Security	Fixed
3	Medium	External controllable retry in the `retrySwapinAndExec` function	DeFi Security	Fixed
4	-	Check the wrapped native token address	Recommendation	Acknowledged
5	-	Check the calldata length when swapping out with payloads	Recommendation	Fixed
6	-	Incorrect receiver in the execution logic	Notes	Confirmed

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential reentrancy in balance calculation

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `MultichainV7Router` contract, there exist functions which are used to swap out the underlying tokens. Theses functions are refactored to support fee-on-transfer (FoT) tokens, which introduces potential reentrancy problem. The `_anySwapOutUnderlying()` function is subject to the reentrancy problem if the underlying token is ERC-777 compliant [1]. Specifically, this function is invoked by the `anySwapOutUnderlying` function (and the `anySwapOutUnderlyingAndCall` function as well) which does NOT adopt any security mechanism (e.g., reentrancy guard) to mitigate the problem.

A possible exploit is as follows:

1. The attacker calls the `anySwapOutUnderlying` function with an ERC-777 compilant underlying `token` and `amount` A.
2. The `anySwapOutUnderlying` function invokes the `transferFrom` function of the ERC-777 token, which calls back to the implementer indicated by the `msg.sender`. Note that the implementer can also be specified as the `msg.sender` itself.

[1] The ERC-777 standard requires specifying *implementer*s for sending and receiving tokens to perform the *tokensToSend* hook and the *tokensReceived* hook, respectively.

3. The attacker re-enters the `anySwapOutUnderlying` function with the same `token` and another `amount` B. Note that the first transfer is not finished at this moment, so the `old_balance` variable has not been updated accordingly.
4. Because of the reentrancy and the update of the `old_balance` variable afterwards, the contract records the deposit amount as $A + 2B$, while the attacker only deposits $A + B$.

```
function anySwapOutUnderlying(
    address token,
    string memory to,
    uint256 amount,
    uint256 toChainID
) external {
    uint256 recvAmount = _anySwapOutUnderlying(token, amount);
    emit LogAnySwapOut(
        token,
        msg.sender,
        to,
        recvAmount,
        block.chainid,
        toChainID
    );
}
```

Listing 2.1: MultichainV7Router.sol

```
function _anySwapOutUnderlying(address token, uint256 amount)
    internal
    whenNotPaused(Swapout_Paused_ROLE)
    returns (uint256)
{
    address _underlying = IUnderlying(token).underlying();
    require(_underlying != address(0), "MultichainRouter: zero underlying");
    uint256 old_balance = IERC20(_underlying).balanceOf(token);
    IERC20(_underlying).safeTransferFrom(msg.sender, token, amount);
    uint256 new_balance = IERC20(_underlying).balanceOf(token);
    return new_balance > old_balance ? new_balance - old_balance : 0;
}
```

Listing 2.2: MultichainV7Router.sol

Impact The cross-chain swap amount can be manipulated by launching the reentrancy attack if ERC-777 tokens are supported.

Suggestion Add reentrancy guards to prevent the reentrancy problem.

2.1.2 Incorrect logic in the `retrySwapinAndExec` function

Severity Medium

Status Fixed in `Version 3`

Introduced by `Version 2`

Description To fix issue 2.2.1, the project introduced a function named `retrySwapinAndExec` in the `SushiSwapProxy` contract (and the `CurveAaveProxy` contract as well). This function requires the `msg.sender` must be the receiver decoded from the `data`. Then the function will invoke the router's `retrySwapinAndExec` function to prevent the reentrancy problem. However, there are some incorrect logic in the newly implemented functions.

- At line 371 of the `retrySwapinAndExec` function, the `dontExec` implies that the user would prefer not to perform the execution while retrying. In this case, the router would simply withdraw the tokens to the receiver. However, the `if(!dontExec)` statement uses the variable in the opposite way.
- At line 373 of the `retrySwapinAndExec` function, the transferred token is the `MultichainV7ERC20` token. However, if the execution is not performed, the `retrySwapinAndExec` function in the `MultichainV7Router` contract will withdraw the underlying token of the `MultichainV7ERC20` token to the proxy contract. In other words, the use of `token` here should also be `IUnderlying(token).underlying()`.

```
349    function retrySwapinAndExec(
350        address router,
351        string memory swapID,
352        address token,
353        address receiver,
354        uint256 amount,
355        uint256 fromChainID,
356        bytes calldata data,
357        bool dontExec
358    ) external {
359        AnycallInfo memory info = decode_anycall_info(data);
360        require(msg.sender == info.receiver, "forbid call retry");
361        IRetrySwapinAndExec(router).retrySwapinAndExec(
362            swapID,
363            token,
364            receiver,
365            amount,
366            fromChainID,
367            address(this),
368            data,
369            dontExec
370        );
371        if (!dontExec) {
372            // process don't exec situation (eg. return token)
373            IERC20(token).safeTransfer(info.receiver, amount);
374        }
375    }
```

Listing 2.3: SushiSwapProxy.sol (Version2)

```
function retrySwapinAndExec(
    string memory swapID,
    address token,
    address receiver,
    uint256 amount,
    uint256 fromChainID,
    address anycallProxy,
    bytes calldata data,
    bool dontExec
) external nonReentrant {
    require(msg.sender == receiver, "forbid retry swap");
```

```
        require(completedSwapin[swapID], "swap not completed");
        bytes32 retryHash = keccak256(abi.encode(swapID, token, receiver, amount, fromChainID,
            anycallProxy, data));
        require(retryRecords[retryHash], "retry record not exist");
        retryRecords[retryHash] = false;

        if (dontExec) {
            address _underlying = IUnderlying(token).underlying();
            require(_underlying != address(0), "MultichainRouter: zero underlying");
            require(IERC20(_underlying).balanceOf(token) >= amount, "MultichainRouter: retry failed
                ");
            assert(IRouter(token).mint(address(this), amount));
            IUnderlying(token).withdraw(amount, receiver);
        } else {
            _anySwapInUnderlyingAndExec(swapID, token, receiver, amount, fromChainID, anycallProxy,
                data, true);
        }
    }
```

Listing 2.4: MultichainV7Router.sol (Version2)

Impact Incorrect logic will make the retrying process not function as expected.

Suggestion Revise the code accordingly.

2.2 DeFi Security

2.2.1 External controllable retry in the `retrySwapinAndExec` function

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `MultichainV7Router` contract, there is a retrying logic so that if the invocation of the `_anySwapInUnderlyingAndExec` function failed (e.g., due to insufficient funds in the vaults), the user can retry the same request by calling the `retrySwapinAndExec` function. However, the implementation of the `retrySwapinAndExec` function does not have any access control. As a result, anyone knowing the swap content and the swap ID can successfully invoke this function. Normally this would not cause any problem because the swap ID is the hash of the swap content, so it is impossible to forge a non-existent swap.

However, a malicious user can control when the `retrySwapinAndExec` function is called. For example, the execution payload is specified to swap a large amount in a SushiSwap pair. The malicious user can then sandwich this call and make a profit from this large swap by controlling the time to invoke `retrySwapinAndExec` function. The auditors consider this kind of sandwich attacks as real threats because it could be launched in a single transaction and causes the loss of the users. Note that this is only one case of exploiting the controllable retry timing which may also lead to other problems as the call targets are extensible.

```
    function retrySwapinAndExec(
        string memory swapID,
        address token,
```

```
        address receiver,
        uint256 amount,
        uint256 fromChainID,
        address anycallProxy,
        bytes calldata data
    ) external {
        require(completedSwapin[swapID], "swap not completed");
        bytes32 retryHash = keccak256(abi.encode(swapID, token, receiver, amount, fromChainID,
            anycallProxy, data));
        require(retryRecords[retryHash], "retry record not exist");
        retryRecords[retryHash] = false;

        _anySwapInUnderlyingAndExec(swapID, token, receiver, amount, fromChainID, anycallProxy,
            data, true);
    }
```

Listing 2.5: MultichainV7Router.sol

```
    function _anySwapInUnderlyingAndExec(
        string memory swapID,
        address token,
        address receiver,
        uint256 amount,
        uint256 fromChainID,
        address anycallProxy,
        bytes calldata data,
        bool isRetry
    ) internal whenNotPaused(Swapin_Paused_ROLE) whenNotPaused(Exec_Paused_ROLE) nonReentrant {
        require(anycallProxyInfo[anycallProxy].supported, "unsupported ancall proxy");
        completedSwapin[swapID] = true;

        address receiveToken;
```

Listing 2.6: MultichainV7Router.sol

Impact External controllable retry may result in potential risk-free price manipulation attacks.

Suggestion Revise the code accordingly.

Feedback from the Project The severity is not high for there exists the slippoint protection in data and the probablity of retry is not high.

2.3 Additional Recommendation

2.3.1 Check the wrapped native token address

Status Acknowledged

Introduced by `Version 1`

Description The constructor of the `MultichainV7Router` contract checks if the executor address is zero. However, it does not check the `wNATIVE` address (the address for the wrapped native token). There are two reasons to check the `wNATIVE` address:

1. wNATIVE is an immutable state variable, which cannot be modified after the Initialization.
2. wNATIVE will be checked against the zero address for every use. Such checks could be merged into one check in the constructor.

```
constructor(
    address _admin,
    address _mpc,
    address _wNATIVE,
    address _anycallExecutor
) MPCManageable(_mpc) PausableControlWithAdmin(_admin) {
    require(_anycallExecutor != address(0), "zero anycall executor");
    anycallExecutor = _anycallExecutor;
    wNATIVE = _wNATIVE;
}
```

Listing 2.7: MultichainV7Router.sol

```
function _anySwapOutNative(address token)
    internal
    whenNotPaused(Swapout_Paused_ROLE)
    returns (uint256)
{
    require(wNATIVE != address(0), "MultichainRouter: zero wNATIVE");
    require(
        IUnderlying(token).underlying() == wNATIVE,
        "MultichainRouter: underlying is not wNATIVE"
    );
    uint256 old_balance = IERC20(wNATIVE).balanceOf(token);
    IwNATIVE(wNATIVE).deposit{value: msg.value}();
    IERC20(wNATIVE).safeTransfer(token, msg.value);
    uint256 new_balance = IERC20(wNATIVE).balanceOf(token);
    return new_balance > old_balance ? new_balance - old_balance : 0;
}
```

Listing 2.8: MultichainV7Router.sol

Impact N/A

Suggestion Implement the proper checks in the constructor.

Feedback from the Project RouterV7 is a general contract, which may has many instances with different _wNATIVE setting, I mean we allow it being zero address.

2.3.2 Check the calldata length when swapping out with payloads

Status Fixed in Version2

Introduced by Version 1

Description In the MultichainV7Router contract, there is a new mechanism that allows additional execution payloads to be specified along with swapping in and out. In the functions that implement swapping out with payloads (e.g., the anySwapOutAndCall function), there does not exist any check on the length of the calldata (i.e., the execution payload). For the current implementation, all invocations are executed through a whitelisted AnyCallExecutor interface, which means an empty calldata would have no reasonable meaning.

```
function anySwapOutAndCall(
    address token,
    string memory to,
    uint256 amount,
    uint256 toChainID,
    string memory anycallProxy,
    bytes calldata data
) external whenNotPaused(Swapout_Paused_ROLE) whenNotPaused(Call_Paused_ROLE) {
    assert(IRouter(token).burn(msg.sender, amount));
    emit LogAnySwapOutAndCall(
        token,
        msg.sender,
        to,
        amount,
        block.chainid,
        toChainID,
        anycallProxy,
        data
    );
}
```

Listing 2.9: MultichainV7Router.sol

Impact N/A

Suggestion Add sanity checks accordingly.

2.4 Note

2.4.1 Incorrect receiver in the execution logic

Status Confirmed

Introduced by Version 1

Description In the MultichainV7Router contract, the anySwapInAndExec function is implemented to allow users to make specified calls after cross-chain swapping in via the anycallExecutor contract. The receiver parameter passed to the anySwapInAndExec function can be any address, and it is used as the receiver to invoke the IRouter(token).mint function. The mint function allows the receiver to withdraw the underlying tokens from the token contract. Since the anycallProxy is the actual caller of the subsequent calls such as swapping tokens in SushiSwap, if the receiver address is not valid, the anycallProxy will not receive enough tokens to perform the further operations.

```
function anySwapInAndExec(
    string memory swapID,
    address token,
    address receiver,
    uint256 amount,
    uint256 fromChainID,
    address anycallProxy,
```

```
        bytes calldata data
    ) external whenNotPaused(Swapin_Paused_ROLE) whenNotPaused(Exec_Paused_ROLE) checkCompletion(
        swapID) nonReentrant onlyMPC {
        require(anycallProxyInfo[anycallProxy].supported, "unsupported ancall proxy");
        completedSwapin[swapID] = true;

        assert(IRouter(token).mint(receiver, amount));

        bool success;
        bytes memory result;
        try IAnycallExecutor(anycallExecutor).execute(anycallProxy, token, amount, data)
        returns (bool succ, bytes memory res) {
            (success, result) = (succ, res);
        } catch {
        }

        emit LogAnySwapInAndExec(
            swapID,
            token,
            receiver,
            amount,
            fromChainID,
            block.chainid,
            success,
            result
        );
    }
```

Listing 2.10: MultichainV7Router.sol

Impact N/A

Suggestion Revise the code.

Feedback from the Project we'll give warning in docs, and this setting depends on partner, who is responsible for the correctness of parameters.