Security Audit
Report for OmnityPort contract

Date: May 31, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Omnity Network
Target	OmnityPort contract

Version History

Version	Date	Description
1.0	May 31, 2024	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-05-31 02:26:19+0000

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 14 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 code repository of OmnityPort contract [1] of Omnity Network. Please note that this audit scope is limited to the contract `contracts/OmnityPort.sol`.

The `OmnityPort` contract serves as an agent enabling users to bridge their tokens from other settlement blockchains into an EVM-compatible blockchain. The tokens bridged are wrapped into `TokenContract`, an ERC20 token, and can be transported or redeemed anytime.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA 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
OmnityPort contract	Version 1	4ffb88dba3f7828ce2ff44dea5af631ebdb1973b
	Version 2	a0c01cf0e92b9bc6c3ec72cad20c9eea9019d48b

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/octopus-network/omnity-port-solidity

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
* Permission management
* 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 found **five** potential security issues. Besides, we have **two** recommendations and **two** notes.

- Medium Risk: 2
- Low Risk: 3
- Recommendation: 2
- Note: 2

ID	Severity	Description	Category	Status
1	Medium	Failure of updating private variables	Software Security	Fixed
2	Medium	Potential invalid event emissions due to non-compatible tokens	DeFi Security	Confirmed
3	Low	Lack of check on `dstChainId` in the `transportToken` function	DeFi Security	Confirmed
4	Low	Lack of refund in `transportToken` and `redeemToken` functions	DeFi Security	Fixed
5	Low	Potential inconsistent precision due to the automatic scaling of `amount`	DeFi Security	Fixed
6	-	Redundant code	Recommendation	Fixed
7	-	Lack of sanity checks	Recommendation	Fixed
8	-	Potential centralization risks	Note	-
9	-	Potential off-chain risks	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Failure of updating private variables

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract `TokenContract` inherits from the contract `ERC20` and declares private variables `_name` and `_symbol` with the same names as those in `ERC20`. Additionally, `TokenContract` includes two privileged functions, `updateSymbol` and `updateName`, allowing the contract `OmnityPortContract` to update these variables.

However, the scope of these private variables is limited to `TokenContract` itself. As a result, updates to these private variables cannot be accessed through the inherited functions `name()` and `symbol()`, which are not correctly overridden in `TokenContract`.

```
contract TokenContract is ERC20, Ownable {
    uint8 private _decimals;
    string private _name;
    string private _symbol;
```

```
constructor(
    address initialOwner,
    string memory name_,
    string memory symbol_,
    uint8 decimals_
) ERC20(name_, symbol_) Ownable(initialOwner) {
    _decimals = decimals_;
    _symbol = symbol_;
    _name = name_;
}

function updateSymbol(string memory symbol_) public onlyOwner {
    _symbol = symbol_;
}

function updateName(string memory name_) public onlyOwner {
    _name = name_;
}
...
```

Listing 2.1: contracts/OmnityPort.sol

Impact The private variables cannot be updated.

Suggestion Update the correct variables.

2.2 DeFi Security

2.2.1 Potential invalid event emissions due to non-compatible tokens

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description The `_executeDirective` function allows the `owner` to add a token and update its information in the mapping `tokens`. When the `contractAddress` parameter is not empty, it sets this address directly instead of creating a new `TokenContract`.

However, if the specified `contractAddress` is incompatible with the `TokenContract` interface, it may cause unexpected behaviors. For example, if the token contract (e.g., WETH) has a fallback function, the external call `TokenContract(tokens[tokenId].erc20ContractAddr).burn()` in the `transportToken` function can be bypassed, leading to an invalid `TokenTransportRequested` event being emitted. Similar issues also existed in other functions, such as `priviledgedMintToken` and `redeemToken`.

```
function _executeDirective(
    Command command,
    uint256 sequence,
    bytes memory params
) private {
    require(
```

```
            isActive || command == Command.Reinstate,
            "Contract is unactive now!"
        );
        require(
            handledDirectives[sequence] == false,
            "directive had been handled"
        );
        if (command == Command.AddToken) {
            (
                string memory settlementChainId,
                string memory tokenId,
                address contractAddress,
                string memory name,
                string memory symbol,
                uint8 decimals
            ) = abi.decode(
                    params,
                    (string, string, address, string, string, uint8)
                );
            if (contractAddress == address(0)) {
                contractAddress = address(
                    new TokenContract(address(this), name, symbol, decimals)
                );
            }
            TokenInfo memory t = TokenInfo({
                name: name,
                symbol: symbol,
                erc20ContractAddr: contractAddress,
                decimals: decimals,
                settlementChainId: settlementChainId
            });
            tokens[tokenId] = t;
        } else if (command == Command.UpdateFee) {
            (
                FactorType factorType,
                string memory tokenOrChainId,
                uint128 amt
            ) = abi.decode(params, (FactorType, string, uint128));
            if (factorType == FactorType.FeeTokenFactor) {
                feeTokenFactor = amt;
            } else if (factorType == FactorType.TargetChainFactor) {
                targetChainFactor[tokenOrChainId] = amt;
            }
        } else if (command == Command.Suspend) {
            isActive = false;
        } else if (command == Command.Reinstate) {
            isActive = true;
        } else {
            return;
        }
        handledDirectives[sequence] = true;
        lastExecutedSequence = sequence;
        emit DirectiveExecuted(sequence);
```

```
}
```

Listing 2.2: contracts/OmnityPort.sol

Impact Invalid events could be emitted when the token is incompatible with the `TokenContract`.

Suggestion Add checks to avoid invalid token contacts being added.

Feedback from the Project This is by design. If we need to specify the token contract address in the directive, we'll check it manually before apply the action.

2.2.2 Lack of check on `dstChainId` in the `transportToken` function

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The `transportToken` function enables users to bridge their tokens to another chain by burning tokens and emitting a `TokenTransportRequested` event. However, it does not check whether the `dstChainId` is valid, which may result in the potential loss of users' funds if an invalid `dstChainId` is provided.

```
function transportToken(
    string memory dstChainId,
    string memory tokenId,
    string memory receiver,
    uint256 amount,
    string memory memo
) external payable {
    require(amount > 0, "the amount must be more than zero");
    require(
        bytes(receiver).length > 0,
        "the receiver's length can't be zero"
    );
    require(
        msg.value >= calculateFee(dstChainId),
        "Deposit fee is less than transport fee"
    );
    TokenContract(tokens[tokenId].erc20ContractAddr).burn(
        msg.sender,
        amount
    );
    emit TokenTransportRequested(
        dstChainId,
        tokenId,
        receiver,
        amount,
        memo
    );
}
```

Listing 2.3: contracts/OmnityPort.sol

Impact Potential loss of funds when transporting tokens to invalid chains.

Suggestion Implement checks to validate `dstChainId` before processing the token transport.

Feedback from the Project This is by design. Our frontend will check the dstChainId before pass it to the function. If the users call this function manually, they have to take the risk.

2.2.3 Lack of refund in `transportToken` and `redeemToken` functions

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `OmnityPort` contract charges a fee from users in the `transportToken` and `redeemToken` functions. For example, there is a requirement `msg.value >= calculateFee` on line 151 in the function `transportToken`. However, when `msg.value` exceeds the required fee calculated, the surplus funds remain locked in the contract.

```
function swap(address inputToken, uint256 amountIn, uint256 _minOutput, bytes calldata _data)
    internal
    returns (uint256)
{
    SwapData memory swapData = bytesToSwapData(_data);

    IERC20(inputToken).forceApprove(swapData.addressToApprove, amountIn);

    uint256 returnAmount;
    (bool success, bytes memory responseData) = swapData.addressToCall.call{ value: swapData.
        value }(swapData.data);
    if (success) {
        returnAmount = abi.decode(responseData, (uint256));
    } else {
        revert Errors.DexSwapFailed();
    }

    if (returnAmount < _minOutput) {
        revert Errors.ReceivedLessThanMinOutput();
    }
    return returnAmount;
}
```

Listing 2.4: contracts/OmnityPort.sol

Impact Surplus fees paid by users are locked in the `OmnityPort` contract.

Suggestion Add refund logic accordingly.

2.2.4 Potential inconsistent precision due to the automatic scaling of `amount`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the current contract `TokenContract`, the `amount` for minting or burning tokens is automatically scaled to match the token decimals. This scaling can lead to precision issues as the smallest unit that can be minted or burned is $10^{d}ecimals$, potentially resulting in mismatches or inaccuracies.

```
function mint(address receiver, uint256 amount) public onlyOwner {
    _mint(receiver, amount * 10 ** (uint256(decimals())));
}

function burn(address owner, uint256 amount) public onlyOwner {
    _burn(owner, amount * 10 ** (uint256(decimals())));
}
```

Listing 2.5: contracts/OmnityPort.sol

Impact The `mint` and `burn` functions cannot handle amounts with precision.

Suggestion Revise the code accordingly.

2.3 Additional Recommendation

2.3.1 Redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There is redundant code in the `OmnityPort` contract. For example, the `AddChain` directive is defined in the `Command` type but is not handled in the `_executeDirective` function. Additionally, variables such as `omnityChainId` are declared but not used. Removing these unused codes will improve the readability of the source code.

Suggestion Remove unused codes accordingly or implement the necessary logic to utilize them correctly.

2.3.2 Lack of sanity checks

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In `OmnityPort`, some functions lack sufficient sanity checks for key parameters. For example, the `_chainKeyAddress` parameter is not checked for a non-zero value in the constructor. Since it cannot be changed after creation, an accidental assignment of a zero address would be irreversible. Similar issues also exist in functions such as `collectFee`.

Suggestion Implement sanity checks accordingly.

2.4 Note

2.4.1 Potential centralization risks

Description The `OmnityPort` contract contains several privileged functions that allow actions such as issuing tokens or executing directives. This introduces a risk of centralization, as the privileged accounts can significantly impact the functionality and security of the protocol.

Feedback from the Project This is by design. The port is controlled by Omnity route canister.

2.4.2 Potential off-chain risks

Description There are features implemented off-chain, such as decoding and handling specified events in the `OmnityPort` contract. These off-chain features may impact the bridge progress. It is important to implement these features correctly to ensure the integrity and functionality.

Feedback from the Project This is by design.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS