SMART CONTRACT AUDIT REPORT

for

SGNv2 Message

Prepared By: Yiqun Chen

Hangzhou, China

February 20, 2022

Document Properties

Client	Celer Network
Title	Smart Contract Audit Report
Target	SGNv2 Message
Version	1.0
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 20, 2022	Shulin Bie	Final Release
1.0-rc	February 19, 2022	Shulin Bie	Release Candidate

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 4
1.1 About SGNv2 Message . . . 4
1.2 About PeckShield . . . 5
1.3 Methodology . . . 5
1.4 Disclaimer . . . 7

2 Findings 9
2.1 Summary . . . 9
2.2 Key Findings . . . 10

3 Detailed Results 11
3.1 Improper Logic Of MessageBusSender::withdrawFee() . . . 11
3.2 Improper Logic Of MessageBusReceiver::verifyTransfer() . . . 12
3.3 Meaningful Events For Important State Changes . . . 14
3.4 Trust Issue Of Admin Keys . . . 15

4 Conclusion 17

References 18

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the SGNv2 Message, 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 SGNv2 Message

Celer Network is an Internet-scale, trust-free, and privacy-preserving platform where everyone can quickly build, operate, and use highly scalable decentralized applications. Celer Network provides unprecedented performance and flexibility through innovation in off-chain scaling techniques and incentive-aligned crypto-economics. SGNv2 is an important part of Celer Network, which supports cross-chain transaction between different blockchains. In particular, the audited Message provides message cross-chain service, which brings more flexibility for cross-chain transactions.

Table 1.1: Basic Information of SGNv2 Message

Item	Description
Target	SGNv2 Message
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	February 20, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Please note that this audit only covers the contracts in the contracts/message directory and the contracts/message/apps sub-directory is out of the audit scope.

- https://github.com/celer-network/sgn-v2-contracts.git (fbd4992)

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

- https://github.com/celer-network/sgn-v2-contracts.git (71a1955)

1.2 About PeckShield

PeckShield Inc. [9] 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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- 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.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], 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 `SGNv2 Message` implementation. 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	2	■■
Medium	0	
Low	1	■
Informational	1	■
Total	4	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others 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 high-severity vulnerabilities, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key SGNv2 Message Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Improper Logic Of MessageBusSender::withdrawFee()	Business Logic	Fixed
PVE-002	High	Improper Logic Of MessageBusReceiver::verifyTransfer()	Business Logic	Fixed
PVE-003	Informational	Meaningful Events For Important State Changes	Coding Practices	Fixed
PVE-004	Low	Trust Issue Of Admin Keys	Security Features	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 Improper Logic Of MessageBusSender::withdrawFee()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: MessageBusSender
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

By design, the MessageBusSender contract is one of the core entries of the SGNv2 Message, which supports arbitrary message transfer along with carrying out cross-chain assets transactions. Additionally, the transaction fee is related to the length of the message. In particular, one entry routine, i.e., withdrawFee(), is designed to withdraw the fee to the specified account. While examining its logic, we notice there is an improper implementation that needs to be improved.

To elaborate, we show below the related code snippet of the MessageBusSender contract. In the withdrawFee() routine, the input _cumulativeFee parameter stores the cumulative fees of the account (specified by the input _account parameter), and the withdrawnFees[_account] storage variable stores the fees that the _account has withdrawn. The statement (i.e., uint256 amount = _cumulativeFee - withdrawnFees[_account]) (line 92) is executed to calculate the fees that the _account can withdraw this time. However, we notice the withdrawnFees[_account] storage variable is not updated anywhere in the routine, which results in that the _account may withdraw unexpected fees. Given this, we suggest to update the withdrawnFees[_account] variable as below: withdrawnFees[_account] += amount (line 93).

```
83      function withdrawFee(
84          address _account,
85          uint256 _cumulativeFee,
86          bytes[] calldata _sigs,
87          address[] calldata _signers,
88          uint256[] calldata _powers
```

```
    ) external {
        bytes32 domain = keccak256(abi.encodePacked(block.chainid, address(this), "
            withdrawFee"));
        sigsVerifier.verifySigs(abi.encodePacked(domain, _account, _cumulativeFee),
            _sigs, _signers, _powers);
        uint256 amount = _cumulativeFee - withdrawnFees[_account];
        require(amount > 0, "No new amount to withdraw");
        (bool sent, ) = _account.call{value: amount, gas: 50000}("");
        require(sent, "failed to withdraw fee");
    }
```

Listing 3.1: `MessageBusSender::withdrawFee()`

Recommendation Correct the implementation of the `withdrawFee()` routine by properly updating the `withdrawnFees[_account]` storage variable.

Status The issue has been addressed by the following commit: `ce081e8`.

3.2 Improper Logic Of MessageBusReceiver::verifyTransfer()

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: High
- Target: `MessageBusReceiver`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned in Section 3.1, `SGNv2 Message` supports arbitrary message transfer along with carrying out cross-chain assets transactions. The `MessageBusReceiver` contract is designed to deal with the cross-chain message on the destination chain. By design, only when the cross-chain assets transfer has finished on the destination chain, the corresponding message can be dealt with. In particular, one internal routine, i.e., `MessageBusReceiver::verifyTransfer()`, is called inside the `executeMessageWithTransfer()` routine to verify whether the cross-chain assets transfer associated with the message has finished. While examining its logic, we notice there is an improper implementation that needs to be improved.

To elaborate, we show below the related code snippet of the `MessageBusReceiver` contract. We notice the `verifyTransfer()` routine is compatible with three kinds of cross-chain bridges (The bridge-related contracts are stored in `liquidityBridge`, `pegBridge/pegVault`, and `pegBridgeV2/pegVaultV2`.) that the `SGNv2` protocol supports. However, it comes to our attention that the `pegBridge/pegVault` rather than `pegBridgeV2/pegVaultV2` are incorrectly used to support the last kind of cross-chain bridge, which directly undermines the assumption of the design.

```
    function verifyTransfer(TransferInfo calldata _transfer) private view returns (bytes32) {
        bytes32 transferId;
        address bridgeAddr;
        if (_transfer.t == TransferType.LqSend) {
            ...
        } else if (_transfer.t == TransferType.LqWithdraw) {
            ...
        } else if (_transfer.t == TransferType.PegMint  _transfer.t == TransferType.PegWithdraw) {
            ...
        } else if (_transfer.t == TransferType.PegMintV2  _transfer.t == TransferType.PegWithdrawV2) {
            if (_transfer.t == TransferType.PegMintV2) {
                bridgeAddr = pegBridge;
            } else {
                // TransferType.PegWithdrawV2
                bridgeAddr = pegVault;
            }
            transferId = keccak256(
                abi.encodePacked(
                    _transfer.receiver,
                    _transfer.token,
                    _transfer.amount,
                    _transfer.sender,
                    _transfer.srcChainId,
                    _transfer.refId,
                    bridgeAddr
                )
            );
            if (_transfer.t == TransferType.PegMintV2) {
                require(IPeggedTokenBridge(bridgeAddr).records(transferId) == true, "mint record not exist");
            } else {
                // TransferType.PegWithdrawV2
                require(IOriginalTokenVault(bridgeAddr).records(transferId) == true, "withdraw record not exist");
            }
        }
        return keccak256(abi.encodePacked(MsgType.MessageWithTransfer, bridgeAddr, transferId));
    }
```

Listing 3.2: `MessageBusReceiver::verifyTransfer()`

Recommendation Correct the implementation of the `verifyTransfer()` routine.

Status The issue has been addressed by the following commit: `32410b8`.

3.3 Meaningful Events For Important State Changes

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

While examining the events that reflect the protocol dynamics, we notice there are several privileged routines that lack meaningful events to reflect their changes. In the following, we show several representative routines.

```
    function setMessageBus(address _messageBus) public onlyOwner {
        messageBus = _messageBus;
    }
```

Listing 3.3: MessageBusAddress::setMessageBus()

```
    function setLiquidityBridge(address _addr) public onlyOwner {
        liquidityBridge = _addr;
    }

    function setPegBridge(address _addr) public onlyOwner {
        pegBridge = _addr;
    }

    function setPegVault(address _addr) public onlyOwner {
        pegVault = _addr;
    }

    function setPegBridgeV2(address _addr) public onlyOwner {
        pegBridgeV2 = _addr;
    }

    function setPegVaultV2(address _addr) public onlyOwner {
        pegVaultV2 = _addr;
    }
```

Listing 3.4: MessageBusReceiver

```
    function setFeePerByte(uint256 _fee) external onlyOwner {
        feePerByte = _fee;
    }

    function setFeeBase(uint256 _fee) external onlyOwner {
        feeBase = _fee;
    }
```

Listing 3.5: `MessageBusSender::setFeePerByte()&&setFeeBase()`

With that, we suggest to emit meaningful events in these privileged routines. Also, the key event information is better `indexed`. Note each emitted event is represented as a topic that usually consists of the signature (from a `keccak256` hash) of the event name and the types (`uint256`, `string`, etc.) of its parameters. Each indexed type will be treated like an additional topic. If an argument is not indexed, it will be attached as data (instead of a separate topic). Considering that the key information is typically queried, it is better treated as a topic, hence the need of being `indexed`.

Recommendation Properly emit the above-mentioned events with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status The issue has been addressed by the following commit: `ce081e8`.

3.4 Trust Issue Of Admin Keys

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

Description

In the `SGNv2 Message` implementation, there is a privileged account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function setLiquidityBridge(address _addr) public onlyOwner {
        liquidityBridge = _addr;
    }

    function setPegBridge(address _addr) public onlyOwner {
        pegBridge = _addr;
    }

    function setPegVault(address _addr) public onlyOwner {
```

```
        pegVault = _addr;
    }

    function setPegBridgeV2(address _addr) public onlyOwner {
        pegBridgeV2 = _addr;
    }

    function setPegVaultV2(address _addr) public onlyOwner {
        pegVaultV2 = _addr;
    }
```

Listing 3.6: `MessageBusReceiver`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed 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

4 Conclusion

In this audit, we have analyzed the `SGNv2 Message` design and implementation. `Celer Network` is an Internet-scale, trust-free, and privacy-preserving platform where everyone can quickly build, operate, and use highly scalable decentralized applications. `Celer Network` provides unprecedented performance and flexibility through innovation in off-chain scaling techniques and incentive-aligned crypto-economics. `SGNv2` is an important part of `Celer Network`, which supports cross-chain transaction between different blockchains. In particular, the audited `Message` provides message cross-chain service, which brings more flexibility for cross-chain transaction. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.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: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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

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

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

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