



# SMART CONTRACT AUDIT REPORT

for

## SGN V2



Prepared By: Yiqun Chen

Hangzhou, China  
November 15, 2021

## Document Properties

Client	Celer Network
Title	Smart Contract Audit Report
Target	SGN V2
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	November 15, 2021	Shulin Bie	Final Release
1.0-rc	October 18, 2021	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

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	About SGN V2 . . . . .	4
1.2	About PeckShield . . . . .	5
1.3	Methodology . . . . .	5
1.4	Disclaimer . . . . .	7
<b>2</b>	<b>Findings</b>	<b>9</b>
2.1	Summary . . . . .	9
2.2	Key Findings . . . . .	10
<b>3</b>	<b>Detailed Results</b>	<b>11</b>
3.1	Improved Validation Of Function Arguments . . . . .	11
3.2	Improper Logic In Viewer::getMinValidatorTokens() . . . . .	12
3.3	Incompatibility With Deflationary/Rebasing Tokens . . . . .	13
3.4	Trust Issue Of Admin Keys . . . . .	15
3.5	Improper Logic In Staking::getDelegatorInfo() . . . . .	16
3.6	Suggested Fine-Grained Risk Control Of Transfer Volume . . . . .	17
<b>4</b>	<b>Conclusion</b>	<b>20</b>
	<b>References</b>	<b>21</b>

# 1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the SGN V2 protocol in Celer Network, 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 SGN V2

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. SGN V2, as an important part of Celer Network, supports cross-chain transaction between different blockchains, which enriches the Celer Network ecosystem and also presents a unique contribution to current DeFi ecosystem.

The basic information of SGN V2 is as follows:

Table 1.1: Basic Information of SGN V2

Item	Description
Target	SGN V2
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	November 15, 2021

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

- <https://github.com/celer-network/sgn-v2-contracts.git> (7648031)

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> (a4f213f)

## 1.2 About PeckShield

PeckShield Inc. [10] 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](mailto:contact@peckshield.com)).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

- 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) [8], 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
<b>Configuration</b>	Weaknesses in this category are typically introduced during the configuration of the software.
<b>Data Processing Issues</b>	Weaknesses in this category are typically found in functionality that processes data.
<b>Numeric Errors</b>	Weaknesses in this category are related to improper calculation or conversion of numbers.
<b>Security Features</b>	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
<b>Time and State</b>	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.
<b>Error Conditions, Return Values, Status Codes</b>	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.
<b>Resource Management</b>	Weaknesses in this category are related to improper management of system resources.
<b>Behavioral Issues</b>	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
<b>Business Logics</b>	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.
<b>Initialization and Cleanup</b>	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
<b>Arguments and Parameters</b>	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
<b>Expression Issues</b>	Weaknesses in this category are related to incorrectly written expressions within code.
<b>Coding Practices</b>	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 `SGM v2` 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	0	
Medium	2	
Low	2	
Informational	1	
Undetermined	1	
Total	6	

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

## 2.2 Key Findings

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

Table 2.1: Key SGN V2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	<a href="#">Improved Validation Of Function Arguments</a>	Coding Practices	Fixed
PVE-002	Low	<a href="#">Improper Logic In Viewer::getMinValidatorTokens()</a>	Business Logic	Fixed
PVE-003	Low	<a href="#">Incompatibility With Deflationary/Rebasing Tokens</a>	Business Logic	Confirmed
PVE-004	Medium	<a href="#">Trust Issue Of Admin Keys</a>	Security Features	Confirmed
PVE-005	Medium	<a href="#">Improper Logic In Staking::getDelegatorInfo()</a>	Business Logic	Fixed
PVE-006	Undetermined	<a href="#">Suggested Fine-Grained Risk Control Of Transfer Volume</a>	Security Features	Mitigated

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 Validation Of Function Arguments

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Bridge
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

#### Description

According to the SGN v2 design, the Bridge contract is designed to be the main entry for cross-chain transactions. In particular, one routine, i.e., `setMinSend()`, is designed to update the minimum amount of cross-chain transactions per asset. While examining the logic of the `setMinSend()` routine, we observe the current implementation can be improved.

To elaborate, we show below the related code snippet of the contract. In the `setMinSend()` function, we notice it has the inherent assumption on the same length of the given two arrays, i.e., `tokens` and `minsend`. However, this is not enforced inside the `setMinSend()` function. We believe it is necessary to validate the length of the two arrays.

```
113 function setMinSend(address[] calldata tokens, uint256[] calldata minsend) external  
    onlyOwner {  
114     for (uint256 i = 0; i < tokens.length; i++) {  
115         minSend[tokens[i]] = minsend[i];  
116     }  
117 }
```

Listing 3.1: Bridge::setMinSend()

**Recommendation** Validate the length of the two arrays in the `setMinSend()` routine.

**Status** The issue has been addressed by the following commit: 1a05fe9.

## 3.2 Improper Logic In Viewer::getMinValidatorTokens()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Viewer
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

### Description

In the SGN V2 protocol, the `Viewer` contract is designed to check the validators-related information. In particular, one routine, i.e., `getMinValidatorTokens()`, is designed to retrieve the minimum amount of staking pool from all bonded validators. While examining the logic of the `getMinValidatorTokens()` routine, we notice an improper implementation that can be improved.

To elaborate, we show below the related code snippet of the contract. The `getMinValidatorTokens()` function implements a simple traversal lookup mechanism to find the minimum amount of staking pool. However, we notice the `validators` list is iterated from index 1 rather than index 0. If we assume the index 0 of the `validators` list stores the minimum amount of staking pool, the function will return an incorrect value. With that, it is necessary to iterate the `validators` list from index 0.

```

100     function getMinValidatorTokens() public view returns (uint256) {
101         uint256 bondedValNum = staking.getBondedValidatorNum();
102         if (bondedValNum < staking.params(dt.ParamName.MaxBondedValidators)) {
103             return 0;
104         }
105         uint256 minTokens = dt.MAX_INT;
106         for (uint256 i = 1; i < bondedValNum; i++) {
107             uint256 tokens = staking.getValidatorTokens(staking.bondedValAddrs(i));
108             if (tokens < minTokens) {
109                 minTokens = tokens;
110                 if (minTokens == 0) {
111                     return 0;
112                 }
113             }
114         }
115         return minTokens;
116     }

```

Listing 3.2: `Viewer::getMinValidatorTokens()`

**Recommendation** Improve the logic of the `getMinValidatorTokens()` routine as above-mentioned.

**Status** The issue has been addressed by the following commit: `1a05fe9`.

### 3.3 Incompatibility With Deflationary/Rebasing Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Bridge
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

#### Description

In the SGN v2 implementation, the Bridge contract is designed to transfer assets between blockchains. In particular, one routine, i.e., `send()`, is designed to lock a certain amount (specified by the input `_amount` parameter) of ERC20 token (specified by the input `_token` parameter) in the source chain. Meanwhile, the equal amount of corresponding tokens on the destination chain will be transferred to the recipient with the calling of `relay()` function.

```

50     function send(
51         address _receiver,
52         address _token,
53         uint256 _amount,
54         uint64 _dstChainId,
55         uint64 _nonce,
56         uint32 _maxSlippage // slippage * 1M, eg. 0.5% -> 5000
57     ) external nonReentrant {
58         require(_amount > minSend[_token], "amount too small");
59         require(_maxSlippage > mams, "max slippage too small");
60         bytes32 transferId = keccak256(
61             // uint64(block.chainid) for consistency as entire system uses uint64 for
62             // chain id
63             abi.encodePacked(msg.sender, _receiver, _token, _amount, _dstChainId, _nonce
64                 , uint64(block.chainid))
65         );
66         require(transfers[transferId] == false, "transfer exists");
67         transfers[transferId] = true;
68         IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
69
70         emit Send(transferId, msg.sender, _receiver, _token, _amount, _dstChainId,
71             _nonce, _maxSlippage);
72     }
73
74     function relay(
75         bytes calldata _relayRequest,
76         bytes[] calldata _sigs,
77         address[] calldata _signers,
78         uint256[] calldata _powers
79     ) external {
80         verifySigs(_relayRequest, _sigs, _signers, _powers);
81         PbBridge.Relay memory request = PbBridge.decRelay(_relayRequest);

```

```

79     require(request.dstChainId == block.chainid, "dst chainId not match");

81     bytes32 transferId = keccak256(
82         abi.encodePacked(
83             request.sender,
84             request.receiver,
85             request.token,
86             request.amount,
87             request.srcChainId,
88             request.dstChainId,
89             request.srcTransferId
90         )
91     );
92     require(transfers[transferId] == false, "transfer exists");
93     transfers[transferId] = true;
94     if (request.token == nativeWrap) {
95         // withdraw then transfer native to receiver
96         IWETH(nativeWrap).withdraw(request.amount);
97         payable(request.receiver).transfer(request.amount);
98     } else {
99         IERC20(request.token).safeTransfer(request.receiver, request.amount);
100    }

102    emit Relay(
103        transferId,
104        request.sender,
105        request.receiver,
106        request.token,
107        request.amount,
108        request.srcChainId,
109        request.srcTransferId
110    );
111 }

```

Listing 3.3: Bridge::send()&amp;&amp;relay()

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 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 cross-chain trade related routines. In other words, the above operations, such as `send()/relay()`, may introduce unexpected balance loss locked in the Bridge contract because the actual amount of ERC20 token transferred into the Bridge contract is less than the deposit amount in this situation.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the Bridge before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary

to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into Bridge. In the SGN v2 protocol, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., USDT) that may have control switches that can be dynamically exercised to suddenly become one.

**Recommendation** If current codebase needs to support possible deflationary tokens, it is better 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 by the team. There is no need for deflationary/rebasing tokens support.

### 3.4 Trust Issue Of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

#### Description

In the SGN v2 protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., cross-chain funds management). In the following, we show the representative functions potentially affected by the privilege of the `owner` account.

```

94     function drainToken(address _token, uint256 _amount) external whenPaused onlyOwner {
95         IERC20(_token).safeTransfer(msg.sender, _amount);
96     }

```

Listing 3.4: `FarmingRewards::drainToken()`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the `owner` is not governed by a DAO-like structure. Note that a compromised account would allow the attacker to transfer all locked funds out of the contracts, which directly undermines the assumption of the SGN v2 design.

Note that other routines, i.e., `SGN::drainToken()`, `Staking::drainToken()` and `StakingReward::drainToken()`, share the same issue.

**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** This issue has been confirmed by the team.

### 3.5 Improper Logic In Staking::getDelegatorInfo()

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Staking
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

#### Description

According to the SGN v2 design, the Staking contract is designed to be the main entry for the Celer Token holders to delegate/undelegate their Celer Token to the validators. After that, the validators can participate in the SGN v2 protocol governance on behalf of the holders. The validators storage mapping in the Staking contract stores the relationship between the validator and delegator, including the delegation/undelegation information. In particular, one routine, i.e., `getDelegatorInfo()`, is designed to retrieve the information based on the validator (specified by the input `_valAddr` parameter) and delegator (specified by the input `_delAddr` parameter). While examining the logic of the routine, we notice an improper logic that can be improved.

To elaborate, we show below the related code snippet of the contract. In the `getDelegatorInfo()` function, the specified `undelegations.queue` is iterated over to calculate the delegator's uncompleted undelegation shares (lines 618-624). For this purpose, the traversal should be started from index `undelegations.head` (pointing to the first uncompleted undelegation) of the `undelegations.queue`. However, we notice the `undelegationShares` is accumulated from index 0 of the `undelegations.queue`. With that, we suggest to improve the `getDelegatorInfo()` function as below: `undelegationShares += undelegations[i].shares` (line 623).

```

613     function getDelegatorInfo(address _valAddr, address _delAddr) public view returns (
        dt.DelegatorInfo memory) {
614         dt.Validator storage validator = validators[_valAddr];
615         dt.Delegator storage d = validator.delegators[_delAddr];
616         uint256 tokens = _shareToTokens(d.shares, validator.tokens, validator.shares);
617
618         uint256 undelegationShares;
619         uint256 len = d.undelegations.tail - d.undelegations.head;

```



```

620     dt.Undelegation[] memory undelegations = new dt.Undelegation[](len);
621     for (uint256 i = 0; i < len; i++) {
622         undelegations[i] = d.undelegations.queue[i + d.undelegations.head];
623         undelegationShares += d.undelegations.queue[i].shares;
624     }
625     uint256 undelegationTokens = _shareToTokens(
626         undelegationShares,
627         validator.undelegationTokens,
628         validator.undelegationShares
629     );
630
631     return dt.DelegatorInfo(_valAddr, tokens, d.shares, undelegationTokens,
632         undelegations);
633 }

```

Listing 3.5: Staking::getDelegatorInfo()

**Recommendation** Improve the logic of the `getDelegatorInfo()` routine as above-mentioned.

**Status** The issue has been addressed by the following commit: `1a05fe9`.

## 3.6 Suggested Fine-Grained Risk Control Of Transfer Volume

- ID: PVE-006
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: Bridge
- Category: Security Features [5]
- CWE subcategory: CWE-654 [3]

### Description

According to the SGN v2 design, the Bridge contract will likely accumulate a huge amount of assets with the increased popularity of cross-chain transactions. While examining the implementation of the Bridge, we notice there is no risk control based on the requested transfer amount, including but not limited to daily transfer volume restriction and per-transaction transfer volume restriction. This is reasonable under the assumption that the protocol will always work well without any vulnerability and the validator keys are always properly managed. In the following, we take the `Bridge::send()/relay()` routines to elaborate our suggestion.

Specifically, we show below the related code snippet of the Bridge contract. According to the SGN v2 design, when the `send()` function is called on the source chain, the `relay()` function on the destination chain will be called subsequently to transfer a certain amount of corresponding tokens to the recipient, in order to reach the cross-chain transfer purpose. In the `relay()` function, we notice the `Signers::verifySigs()` function is called (line 77) to ensure the transaction has been signed by

the validators owning at least  $\frac{2}{3}$  of the decision-making powers. Moreover, we notice the SGN v2 protocol is well designed in that the validators are selected by the Celer Token holders and updated timely based on the delegation/undelegation of the holders. However, we cannot guarantee that the mechanism will always work faultlessly. If we assume the validators are hijacked or leaked, all the assets locked up in the Bridge contract will be stolen. To mitigate, we suggest to add fine-grained risk controls based on the requested transfer volume. A guarded launch process is also highly recommended.

```

50     function send(
51         address _receiver,
52         address _token,
53         uint256 _amount,
54         uint64 _dstChainId,
55         uint64 _nonce,
56         uint32 _maxSlippage // slippage * 1M, eg. 0.5% -> 5000
57     ) external nonReentrant {
58         require(_amount > minSend[_token], "amount too small");
59         require(_maxSlippage > mams, "max slippage too small");
60         bytes32 transferId = keccak256(
61             // uint64(block.chainid) for consistency as entire system uses uint64 for
62             // chain id
63             abi.encodePacked(msg.sender, _receiver, _token, _amount, _dstChainId, _nonce
64                 , uint64(block.chainid))
65         );
66         require(transfers[transferId] == false, "transfer exists");
67         transfers[transferId] = true;
68         IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
69         emit Send(transferId, msg.sender, _receiver, _token, _amount, _dstChainId,
70             _nonce, _maxSlippage);
71     }
72
73     function relay(
74         bytes calldata _relayRequest,
75         bytes[] calldata _sigs,
76         address[] calldata _signers,
77         uint256[] calldata _powers
78     ) external {
79         verifySigs(_relayRequest, _sigs, _signers, _powers);
80         PbBridge.Relay memory request = PbBridge.decRelay(_relayRequest);
81         require(request.dstChainId == block.chainid, "dst chainId not match");
82
83         bytes32 transferId = keccak256(
84             abi.encodePacked(
85                 request.sender,
86                 request.receiver,
87                 request.token,
88                 request.amount,
89                 request.srcChainId,
90                 request.dstChainId,

```

```

89         request.srcTransferId
90     )
91 );
92 require(transfers[transferId] == false, "transfer exists");
93 transfers[transferId] = true;
94 if (request.token == nativeWrap) {
95     // withdraw then transfer native to receiver
96     IWETH(nativeWrap).withdraw(request.amount);
97     payable(request.receiver).transfer(request.amount);
98 } else {
99     IERC20(request.token).safeTransfer(request.receiver, request.amount);
100 }
101
102 emit Relay(
103     transferId,
104     request.sender,
105     request.receiver,
106     request.token,
107     request.amount,
108     request.srcChainId,
109     request.srcTransferId
110 );
111 }

```

Listing 3.6: Bridge::send() &amp;&amp; relay()

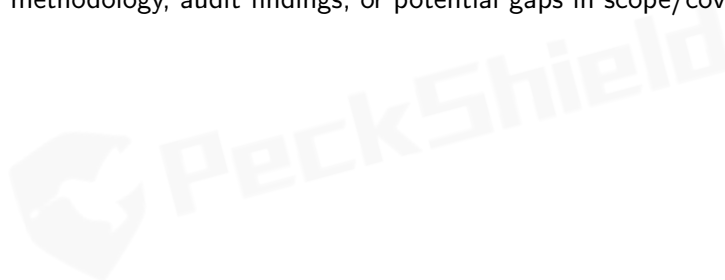
**Recommendation** We suggest to add fine-grained risk controls, including but not limited to daily transfer volume restriction and per transaction transfer volume restriction.

**Status** The issue has been mitigated by the following commits: 601e68b && bad4774 && 943495c.

## 4 | Conclusion

In this audit, we have analyzed the SGN v2 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. SGN v2, as an important part of Celer Network, supports cross-chain transaction between different blockchains, which enriches the Celer Network ecosystem and also presents a unique contribution to current DeFi ecosystem. 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-1041: Use of Redundant Code. <https://cwe.mitre.org/data/definitions/1041.html>.
- [2] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [3] MITRE. CWE-654: Reliance on a Single Factor in a Security Decision. <https://cwe.mitre.org/data/definitions/654.html>.
- [4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [5] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [6] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [7] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [8] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [9] OWASP. Risk Rating Methodology. [https://www.owasp.org/index.php/OWASP\\_Risk\\_Rating\\_Methodology](https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology).

[10] PeckShield. PeckShield Inc. <https://www.peckshield.com>.

