SMART CONTRACT AUDIT REPORT

for

SGN V2

Prepared By: Yiqun Chen

Hangzhou, China
February 16, 2022

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	February 16, 2022	Shulin Bie	Final Release
1.0-rc	January 14, 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

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.

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	February 16, 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/pegged/ directory.

- https://github.com/celer-network/sgn-v2-contracts.git (28da916)

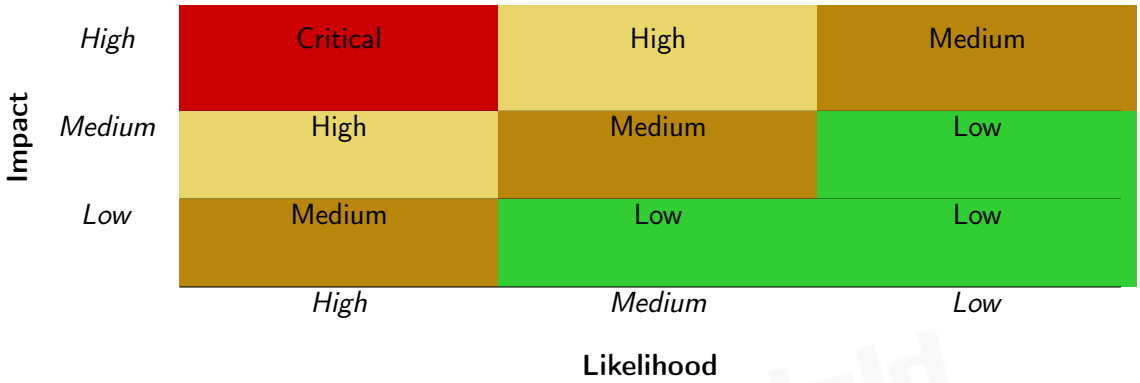
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/celer-network/sgn-v2-contracts.git (8563eea)

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



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.

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

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

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 SGN 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	1	■
Low	2	■■
Informational	0	
Total	3	

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 1 medium-severity vulnerability, and 2 low-severity vulnerabilities.

Table 2.1: Key SGN V2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Accommodation Of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-002	Low	Incompatibility With Deflationary/Rebasing Tokens	Business Logic	Confirmed
PVE-003	Medium	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 Accommodation Of Non-ERC20-Compliant Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MaiBridgeToken`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()`/`transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
194     /**
195     * @dev Approve the passed address to spend the specified amount of tokens on behalf
            of msg.sender.
196     * @param _spender The address which will spend the funds.
197     * @param _value The amount of tokens to be spent.
198     */
199     function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

201         // To change the approve amount you first have to reduce the addresses`
202         //  allowance to zero by calling `approve(_spender, 0)` if it is not
203         //  already 0 to mitigate the race condition described here:
204         //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
205         require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));
```

```
207         allowed[msg.sender][_spender] = _value;
208         Approval(msg.sender, _spender, _value);
209     }
```

Listing 3.1: USDT Token Contract

Because of that, a normal call to approve() with a currently non-zero allowance may fail. In the following, we use the MaiBridgeToken::burn() routine as an example. In this routine, it approves the asset for the maihub contract. To accommodate the specific idiosyncrasy, there is a need to approve() twice (line 64): the first one reduces the allowance to 0; and the second one sets the new allowance.

```
62     function burn(address _from, uint256 _amount) external onlyBridge returns (bool) {
63         IERC20(asset).safeTransferFrom(_from, address(this), _amount);
64         IERC20(asset).approve(address(maihub), _amount);
65         IMaiBridgeHub(maihub).swapOut(address(this), _amount);
66         _burn(address(this), _amount);
67         return true;
68     }
```

Listing 3.2: MaiBridgeToken::burn()

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related approve().

Status The issue has been addressed by the following commit: 8563eea.

3.2 Incompatibility With Deflationary/Rebasing Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: OriginalTokenVault/PeggedTokenBridge
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the SGN V2 protocol, the OriginalTokenVault contract is designed as the core for the cross-chain trade. In particular, one routine, i.e., deposit(), is designed to lock a certain amount (specified by the input _amount parameter) of ERC20 tokens (specified by the input _token parameter) in the source chain. Meanwhile, the equal amount of the corresponding token on the destination chain will be minted as cross-chain trade credits.

```
64      function deposit(
65          address _token,
66          uint256 _amount,
67          uint64 _mintChainId,
68          address _mintAccount,
69          uint64 _nonce
70      ) external nonReentrant whenNotPaused {
71          bytes32 depId = _deposit(_token, _amount, _mintChainId, _mintAccount, _nonce);
72          IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
73          emit Deposited(depId, msg.sender, _token, _amount, _mintChainId, _mintAccount);
74      }
```

Listing 3.3: `OriginalTokenVault::deposit()`

```
49      function mint(
50          bytes calldata _request,
51          bytes[] calldata _sigs,
52          address[] calldata _signers,
53          uint256[] calldata _powers
54      ) external whenNotPaused {
55          bytes32 domain = keccak256(abi.encodePacked(block.chainid, address(this), "Mint"
                ));
56          sigsVerifier.verifySigs(abi.encodePacked(domain, _request), _sigs, _signers,
                _powers);
57          PbPegged.Mint memory request = PbPegged.decMint(_request);
58          bytes32 mintId = keccak256(
59              // len = 20 + 20 + 32 + 20 + 8 + 32 = 132
60              abi.encodePacked(
61                  request.account,
62                  request.token,
63                  request.amount,
64                  request.depositor,
65                  request.refChainId,
66                  request.refId
67              )
68          );
69          require(records[mintId] == false, "record exists");
70          records[mintId] = true;
71          _updateVolume(request.token, request.amount);
72          uint256 delayThreshold = delayThresholds[request.token];
73          if (delayThreshold > 0 && request.amount > delayThreshold) {
74              _addDelayedTransfer(mintId, request.account, request.token, request.amount);
75          } else {
76              IPeggedToken(request.token).mint(request.account, request.amount);
77          }
78          emit Mint(
79              mintId,
80              request.token,
81              request.account,
82              request.amount,
83              request.refChainId,
84              request.refId,
85              request.depositor
```

```
86          );
87      }
```

Listing 3.4: PeggedTokenBridge::mint()

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 deposit()/mint(), may introduce unexpected balance loss locked in the OriginalTokenVault contract because the actual amount of ERC20 token transferred into the OriginalTokenVault 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 OriginalTokenVault 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 OriginalTokenVault . 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.

3.3 Trust Issue Of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the audited part of 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., manage the privileged account, which has capability to mint token arbitrarily). In the following, we show the representative functions potentially affected by the privilege of the `owner` account.

```
30      function mint(address _to, uint256 _amount) external returns (bool) {
31          Supply storage b = bridges[msg.sender];
32          require(b.cap > 0, "invalid caller");
33          b.total += _amount;
34          require(b.total <= b.cap, "exceeds bridge supply cap");
35          _mint(_to, _amount);
36          return true;
37      }
38
39      function burn(address _from, uint256 _amount) external returns (bool) {
40          Supply storage b = bridges[msg.sender];
41          require(b.cap > 0, "invalid caller");
42          b.total -= _amount;
43          _burn(_from, _amount);
44          return true;
45      }
46
47      function updateBridgeSupplyCap(address _bridge, uint256 _cap) external onlyOwner {
48          // cap == 0 means revoking bridge role
49          bridges[_bridge].cap = _cap;
50          emit BridgeSupplyCapUpdated(_bridge, _cap);
51      }
```

Listing 3.5: `MultiBridgeToken`

```
30      function swapBridgeForCanonical(address _bridgeToken, uint256 _amount) external
            returns (uint256) {
31          Supply storage supply = swapSupplies[_bridgeToken];
32          require(supply.cap > 0, "invalid bridge token");
33          require(supply.total + _amount < supply.cap, "exceed swap cap");
34
35          supply.total += _amount;
36          _mint(msg.sender, _amount);
37
```

```
38          // move bridge token from msg.sender to canonical token _amount
39          IERC20(_bridgeToken).safeTransferFrom(msg.sender, address(this), _amount);
40          return _amount;
41      }
42
43      function swapCanonicalForBridge(address _bridgeToken, uint256 _amount) external
            returns (uint256) {
44          Supply storage supply = swapSupplies[_bridgeToken];
45          require(supply.cap > 0, "invalid bridge token");
46
47          supply.total -= _amount;
48          _burn(msg.sender, _amount);
49
50          IERC20(_bridgeToken).safeTransfer(msg.sender, _amount);
51          return _amount;
52      }
53
54      function setBridgeTokenSwapCap(address _bridgeToken, uint256 _swapCap) external
            onlyOwner {
55          swapSupplies[_bridgeToken].cap = _swapCap;
56          emit TokenSwapCapUpdated(_bridgeToken, _swapCap);
57      }
```

Listing 3.6: MintSwapCanonicalToken

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 mint the token arbitrarily, which directly undermines the assumption of the SGN V2 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 This issue has been confirmed by the team.

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. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.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.