CyStack

PUBLIC ACCESS

CYBERSECURITY AUDIT REPORT

Version v1.1

This document details the process and results of the smart contract audit performed by CyStack from 05/11/2024 to 11/11/2024.

Audited for

Demlabs

Audited by

Vietnam CyStack Joint Stock Company

Contents

Disclaimer

Smart Contract Audit only provides findings and recommendations for an exact commitment of a smart contract codebase. The results, hence, are not guaranteed to be accurate outside of the commitment, or after any changes or modifications made to the codebase. The evaluation result does not guarantee the nonexistence of any further findings of security issues.

Time-limited engagements do not allow for a comprehensive evaluation of all security controls, so this audit does not give any warranties on finding all possible security issues of the given smart contract(s). CyStack prioritized the assessment to identify the weakest security controls an attacker would exploit. We recommend Demlabs conduct similar assessments on an annual basis by internal, third-party assessors, or a public bug bounty program to ensure the security of smart contract(s).

This security audit should never be used as investment advice.

Version History

Version	Date	Release notes
1.0	11/11/2024	The first report was sent to the client.
1.1	02/12/2024	Demlabs allowed CyStack to publish the audit report publicly.

Contact Information

Company	Representative	Position	Email address
Demlabs	Dmitriy Gerasimov	General Director	naeper@demlabs.net
CyStack	Nguyen Ngoc Anh	Sales Manager	anhntn@cystack.net

Auditors

Fullname	Role	Email address
Nguyen Huu Trung	Head of Security	trungnh@cystack.net
Nguyen Ba Anh Tuan	Auditor	

Introduction

From 05/11/2024 to 11/11/2024, Demlabs engaged CyStack to evaluate the security posture of the Bridge contracts of their contract system. Our findings and recommendations are detailed here in this initial report.

1.1 Audit Details

Audit Target

In this audit project, CyStack focused on the smart contract for the Bridge contracts of Demlabs

The basic information of is as follows:

Item	Description
Project Name	Bridge contracts
Issuer	Demlabs
Website	https://bridge.cellframe.net/
Platform	Ethereum Smart Contract
Language	Solidity
Codebase	• **Bridge.sol:** https://etherscan.io/address/0x4a831a8ebb160ad025d34a788c99e9320b9ab531#code • **ConsensusOwners.sol:** https://etherscan.io/address/0x7Fb0147936b7E0DDe22e9B6Ec123f87908A9a516#code • **ConsensusNode.sol:** https://etherscan.io/address/0x92cf67713f5ec181FD6999F18af796A6763a225b#code
Commit	N/A
Audit method	Whitebox

The system comprises three primary smart contracts that function together to establish a secure bridge for transferring tokens across different blockchains. The ConsensusNode and ConsensusOwners contracts serve as security layers, requiring multiple approvals from nodes and owners, respectively, before any critical actions can be executed. The Bridge contract facilitates the actual token transfers between chains, but its operations are governed exclusively through these consensus contracts. This setup is akin to a bank vault requiring multiple approvals – "keys" from both security guards (nodes) and bank managers (owners) – before any assets can be moved.

The purpose and key features of each contract are depicted below:

1. Bridge contract:
    - Purpose: Handling the actual token bridging operations.
    - Key features:
        - Locking/unlocking tokens across chains;
        - Interacting with both consensus contracts;
        - Enforcing nodeConsensus role for validating transfers;
        - Enforcing ownersConsensus role for managing liquidity.
2. ConsensusOwners contract:
    - Purpose: Overseeing owner-level governance.
    - Key features:
        - Functioning as a multisig wallet for owners;
        - Requiring multiple owner approvals for transactions;
        - Enforcing a 48-hour grace period for actions;
        - Managing administrative tasks, such as adding/removing owners and adjusting key parameters.
3. ConsensusNode contract:
    - Purpose: Managing validator node operations.
    - Key features:
        - Functioning as an automated consensus for validators;
        - Executing actions immediately, with no grace period;
        - Using nonce for transaction uniqueness;
        - Validating cross-chain transfers;
        - Ideal for frequent, automated operation.

Audit Service Provider

CyStack is a leading security company in Vietnam to build the next generation of cybersecurity solutions to protect businesses against threats from the Internet. CyStack is a member of the Vietnam Information Security Association (VNISA) and Vietnam Alliance for Cybersecurity Products Development.

CyStack's researchers are known as regular speakers at well-known cybersecurity conferences such as BlackHat USA, BlackHat Asia, Xcon, T2FI, etc. and are talented bug hunters who discovered critical vulnerabilities in global products and are acknowledged by their vendors.

1.2 Audit Goals

The audit's focus was to verify that the smart contract system is secure, resilient and working according to its specifications. The audit activities can be grouped into the following three categories:

1. **Security:** Identifying security-related issues within each contract and within the system of contracts.
2. **Sound Architecture:** Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.
3. **Code Correctness and Quality:** A full review of the contract source code. The primary areas of focus include:
    - Correctness
    - Readability
    - Sections of code with high complexity
    - Improving scalability
    - Quantity and quality of test coverage

1.3 Audit Methodology

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

- **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: High, Medium and Low, i.e., H, M and L respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., Critical, Major, Medium, Minor and Informational (Info) as the table below:

Impact				
	High	Critical	Major	Medium
	Medium	Major	Medium	Minor
	Low	Medium	Minor	Informational
		High	*Medium*	*Low*

Likelihood

CyStack firstly analyses the smart contract with open-source and also our own security assessment tools to identify basic bugs related to general smart contracts. These tools include SafeChain, Slither, securify, Mythril, Sūrya, Solgraph, Truffle, Geth, Ganache, Mist, Metamask, solhint, mythx, etc. Among them, SafeChain is a product of CyStack, which initiates fully automated scans on contracts to detect known potential security vulnerabilities in code, monitors, and sends alerts to users when detecting any anomalies in blockchain tokens. Then, our security specialists will verify the tool results manually, make a description and decide the severity for each of them.

After that, we go through a checklist of possible issues that could not be detected with automatic tools, conduct test cases for each and indicate the severity level for the results. If no issues are found after manual analysis, the contract can be considered safe within the test case. Otherwise, if any issues are found, we might further deploy contracts on our private testnet and run tests to confirm the findings. We would additionally build a PoC to demonstrate the possibility of exploitation, if required or necessary.

The standard checklist, which applies for every SCA, strictly follows CyStack's Smart Contract Weakness Classification. This classification is built, strictly following the Smart Contract Weakness Classification Registry (SWC Registry), and is updated frequently according to the most recent emerging threats and new exploit techniques. The checklist of testing according to this classification is shown in Appendix C.

In general, the auditing process focuses on detecting and verifying the existence of the following issues:

- **Data Issues**: Finding bugs in data processing, such as improper names and labels for variables, incorrect inheritance orders and unsafe calculations.
- **Description Issues**: Checking for improper controls of user input that leads to malicious output rendering.
- **Environment Issues**: Inspecting errors related to the environment of some specific Solidity versions.
- **Interaction Issues**: Reviewing the interaction of different smart contracts to locate bugs in handling external calls and controlling the balance and flows of token transfers.
- **Interface Issues**: Investigating the misusage of low-level and token interfaces.
- **Logic Issues**: Testing the code logic and error handlings in the smart contract code base, such as self-DoS attacks, verifying strong randomness, etc.
- **Performance Issues**: Identifying the occurrence of improper byte padding, unused functions and other issues that leads to high gas consumption.
- **Security Issues**: Finding common security issues of the smart contract(s), for example integer overflows, insufficient verification of authenticity, improper use of cryptographic signature, etc.
- **Standard Issues**: Focusing on identifying coding bugs related to general smart contract coding conventions and practices.

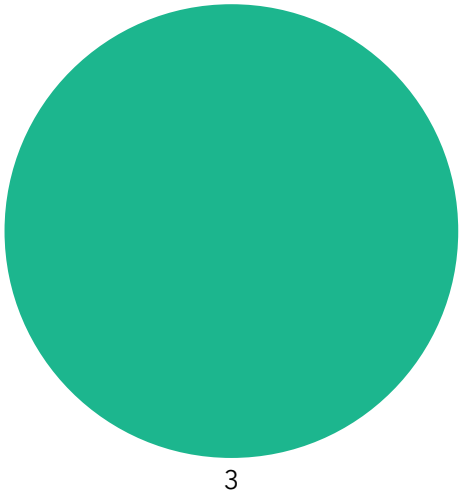
The final report will be sent to the smart contract issuer with an executive summary for overview and detailed results for acts of remediation.

1.4 Audit Scope

Assessment	Target	Type
Original target		
White-box testing	Bridge.sol	Solidity code file
White-box testing	ConsensusOwners.sol	Solidity code file
White-box testing	ConsensusNode.sol	Solidity code file

Executive Summary

Security issues by severity



Legend

- Critical
- Major
- Medium
- Minor
- Info

Security issues by targets

Bridge.sol	1	■
ConsensusOwners.sol	1	■
ConsensusNode.sol	1	■

Security issues by categories

Bussiness Logic (SLD-605)	3	■■■

Table of security issues

ID	Status	Vulnerability	Severity
#cellframe-bridge-001	Open	Lack of input validation in the function bridgeToken	INFO
#cellframe-bridge-002	Open	Lack of input validation in the function MakeProposal	INFO
#cellframe-bridge-003	Open	Lack of input validation in the function addExecProposal	INFO

Recommendations

Based on the results of this smart contract audit, CyStack has the following high-level key recommendations:

Key recommendations	
Issues	CyStack conducted security audit for different contracts in Bridge contracts. No issues with the severity higher than Info had been found. Three issues were found, related to insecure business logic. Currently, all the findings are open.
Recommendations	CyStack recommends Demlabs to evaluate the audit results with several different security audit third parties for the most accurate conclusion.
References	• https://consensys.github.io/smart-contract-best-practices/known_attacks • https://consensys.github.io/smart-contract-best-practices/recommendations/ • https://medium.com/@knownsec404team/ethereum-smart-contract-audit-checklist-ba9d1159b901

Detailed Results

1. Lack of input validation in the function bridgeToken

Issue ID	#cellframe-bridge-001
Category	Logic Issues - Business Logic
Description	Invalid transactions can be processed in the function bridgeToken, since the values of its arguments "token", "value" and "destination_address" are not sanitized before performing the function transferFrom. This may cause inefficient gas consumption when users pass invalid values to these arguments.
Severity	INFO
Location(s)	Bridge.sol: 50-53
Status	Open
Remediation	Add "require" statements to validate the argument values before passing them into the function transferFrom.

Description

The function bridgeToken is declared without strict input validation:

```
…
50      function bridgeToken(address token, uint256  value, bytes3 destination, bytes
        ↪  calldata destination_address) external{
51          IERC20(token).transferFrom(msg.sender, address(this), value);
52          emit BridgeRequest(msg.sender, token, value, networkId, destination,
            ↪  destination_address);
53      }
…
```

Add the following "require" statements to ensure only valid input is passed to the function's arguments:

```
…
50      function bridgeToken(address token, uint256  value, bytes3 destination, bytes
        ↪  calldata destination_address) external{
51          require(token != address(0), "Invalid token address");
52          require(value > 0, "Value must be greater than 0");
53          require(destination_address.length > 0, "Invalid destination address");
54          IERC20(token).transferFrom(msg.sender, address(this), value);
55          emit BridgeRequest(msg.sender, token, value, networkId, destination,
            ↪  destination_address);
56      }
…
```

2. Lack of input validation in the function MakeProposal

Issue ID	#cellframe-bridge-002
Category	Logic Issues - Business Logic
Description	Invalid transactions can be processed in the function MakeProposal, since the values of its argument "_to" is not sanitized before performing the function txToByte. This may cause inefficient gas consumption when users pass invalid values to this argument.
Severity	INFO
Location(s)	ConsensusNode.sol: 77-119
Status	Open
Remediation	Add a "require" statement to validate the argument value before passing it into the function txToByte.

Description

The function MakeProposal is declared without strict input validation:

```
…
77      function MakeProposal(
78          address _to,
79          uint256 _nonce,
80          bytes calldata _data
81      ) external  onlyNode returns(bytes32 txId) {
82
83          txId = txToByte(_to,_data,_nonce);
84
85          //if this consensus already executed, -> do nothing
86          require(!_isExecuted[txId], "proposal already executed");
87
88          //if particular sender node already done this proposal -> do nothing
89          require(!confirmations[txId][msg.sender], "already proposed by you");
90
91          //mark proposal as confirmed by a sender
92          confirmations[txId][msg.sender] = true;
93
94
95          if (eps[txId].uid != 0) //already exists and waiting for confirmation
96          {
97              ExecProposal storage execProposal = eps[txId];
98              execProposal.confirmations++;
99
100             if (execProposal.confirmations >= confirmationsRequired){
101                 callFunction(txId);
102                 emit ProposalExecuted(txId, msg.sender);
103             }
104             else{
105                 emit ProposalAccepted(txId, msg.sender, eps[txId].confirmations);
106             }
107
108
109         } else { //no such proposal, create
110             eps[txId] = ExecProposal({
111                     uid : txId,
112                     to : _to,
113                     data : _data,
114                     nonce : _nonce,
115                     confirmations:1
116                 });
117             emit ProposalAccepted(txId, msg.sender, eps[txId].confirmations);
118         }
119     }
…
```

Add the following "require" statement to ensure only valid input is passed to the function's argument:

```
…
77      function MakeProposal(
78          address _to,
79          uint256 _nonce,
80          bytes calldata _data
81      ) external  onlyNode returns(bytes32 txId) {
82          require(_to != address(0), "Invalid address");
83          txId = txToByte(_to,_data,_nonce);
…           // continues with the original implementation
119     }
…
```

3. Lack of input validation in the function addExecProposal

Issue ID	#cellframe-bridge-003
Category	Logic Issues - Business Logic
Description	Invalid transactions can be processed in the function addExecProposal, since the values of its arguments "_func", "_to" and "_data" are not sanitized before performing the function txToByte. This may cause inefficient gas consumption when users pass invalid values to these arguments.
Severity	INFO
Location(s)	ConsensusNode.sol: 77-107
Status	Open
Remediation	Add "require" statements to validate the argument values before passing them into the function txToByte.

Description

The function addExecProposal is declared without strict input validation:

```
…
77      function addExecProposal(
78          string calldata _func,
79          address _to,
80          bytes calldata _data
81      ) external  onlyOwner returns(bytes32 _txId){
82
83          bytes32 txId = txToByte(_func,_to,_data,block.timestamp);
84          require(!queue[txId],"allready queue");
85
86          queue[txId] = true;
87          confirmations[txId][msg.sender] = true;
88          eps[txId] = ExecProposal({
89                  uid : txId,
90                  to : _to,
91                  func : _func,
92                  data : _data,
93                  timestamp : block.timestamp,
94                  confirmations:1
95                  });
96
97          emit queueTx(
98              msg.sender,
99              _to,
100             _func,
101             _data,
102             block.timestamp,
103             txId
104         );
105
106         return txId;
107     }
…
```

Add the following "require" statements to ensure only valid input is passed to the function's arguments:

```
…
77      function addExecProposal(
78          string calldata _func,
79          address _to,
80          bytes calldata _data
81      ) external  onlyOwner returns(bytes32 _txId){
82          require(_to != address(0), "Invalid target address");
83          require(bytes(_func).length > 0, "Empty function name");
84          require(_data.length > 0, "Empty data");
85          bytes32 txId = txToByte(_func,_to,_data,block.timestamp);
…           // continues with the original implementation
107     }
…
```

Appendices

Appendix A – Security Issue Status Definitions

Status	Definition
Open	The issue has been reported and currently being review by the smart contract developers/issuer.
Unresolved	The issue is acknowledged and planned to be addressed in future. At the time of the corresponding report version, the issue has not been fixed.
Resolved	The issue is acknowledged and has been fully fixed by the smart contract developers/issuer.
Rejected	The issue is considered to have no security implications or to make only little security impacts, so it is not planned to be addressed and won't be fixed.

Appendix B – Severity Explanation

Severity	Definition
CRITICAL	Issues, considered as critical, are straightforwardly exploitable bugs and security vulnerabilities. It is advised to immediately resolve these issues in order to prevent major problems or a full failure during contract system operation.
MAJOR	Major issues are bugs and vulnerabilities, which cannot be exploited directly without certain conditions. It is advised to patch the codebase of the smart contract as soon as possible, since these issues, with a high degree of probability, can cause certain problems for operation of the smart contract or severe security impacts on the system in some way.
MEDIUM	In terms of medium issues, bugs and vulnerabilities exist but cannot be exploited without extra steps such as social engineering. It is advised to form a plan of action and patch after high-priority issues have been resolved.
MINOR	Minor issues are generally objective in nature but do not represent actual bugs or security problems. It is advised to address these issues, unless there is a clear reason not to.
INFO	Issues, regarded as informational (info), possibly relate to “guides for the best practices” or “readability”. Generally, these issues are not actual bugs or vulnerabilities. It is recommended to address these issues, if it makes effective and secure improvements to the smart contract codebase.

Appendix C – Smart Contract Weakness Classification

ID	Name	Description
	Data Issues	
SLD-101	Initialization	Check for Interger Division, Interger Overflow and Underflow, Interger Sign, Interger Truncation and Wrong Operator
SLD-102	Calculation	Check for State Variable Default Visibility, Hidden Built-in Symbols, Hidden State Variables and Incorrect Inheritance Order
SLD-103	Hidden Weaknesses	Check for Unintialized Local/State Variables and Unintialized Storage Variables
	Description Issues	
SLD-201	Output Rendering	Check for RightToLeft Override Control Character
	Environment Issues	
SLD-302	Supporting Software	Check for Deletion of Dynamic Array Elements and Usage of continue Statements In do-while -statements
	Interaction Issues	
SLD-401	Contract Call	Check for Delegatecall to Untrusted Callee, Re-entrancy and Unhandled Exception
SLD-402	Ether Flow	Check for Unprotected Ether Withdrawal, Unexpected Ether Balance, Locked Ether and Pre-sent Ether
	Interface Issues	
SLD-501	Parameter	Check for Externally Controlled Call/delegatecall Data/Address, Hash Collisions with Multiple Variable Length Arguments, Short Address Attack and Signature with Wrong Parameter
SLD-502	Token Interface	Check for Non-standard Token Interface
	Logic Issues	
SLD-601	Assembly Code	Check for Arbitrary Jump with Function Type Variable, Returning Results Using Assembly Code in Constructor and Specifying Function Variable as Any Type

SLD-602	Denial of Service (DoS)	Check for potential DoS due to failed call, by complex fallback function, by gaslimit and by non-exsistent address or malicious contracts
SLD-603	Fairness Problems	Check for Weak Sources of Randomness from Chain Attributes, Usage of Block Values as A Proxy for Time, Results of Contract Execution Affected by Miners and Transaction Order Dependence
SLD-604	Storage	Check for Storage Overlap Attack
SLD-605	Bussiness Logic	Check for Business Logic errors in code
	Performance Issues	
SLD-701	Gas Consumption	Check for Gas Griefing, Byte Padding, Invariants in Loop and Invariants for State Variables that are not declared constant
	Security Issues	
SLD-801	Authority Control	Check for Write to Arbitrary Storage Location, Replay Attack, Suicide Contract, Usage of tx.origin for Authentication/Authorization, Wasteful Contract and Wrong Constructor Name
SLD-802	Privacy	Check for Lack of Proper Signature Verification, Signature Malleability, Non-public Variables that are accessed by public/external functions and Public Data
	Standard Issues	
SLD-901	Maintainability	Check for Implicit Visibility, Non-standard Naming, The Use of Too Many Digits, Unlimited/Outdated Compiler Versions and Usage of Deprecated Built-in Symbols
SLD-902	Programming	Check for Code with No Effect, Message Call with Hardcoded Gas Amount, Presence of Unused Variables, View/Constant Functions that change contract states and Improper Usage of require, assert or revert