SMART CONTRACT AUDIT REPORT

for

Evryhub

Prepared By: Yiqun Chen

PeckShield

November 8, 2021

Document Properties

Client	Evrynet Finance
Title	Smart Contract Audit Report
Target	Evryhub
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 8, 2021	Shulin Bie	Final Release
1.0-rc	October 3, 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

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the `Evryhub`, 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 Evryhub

`Evrynet` is an intelligent financial services platform that enables developers and businesses to build an unlimited number of centralized/decentralized finance (`CeDeFi`) applications. It is interoperable with many of the world's leading blockchains. In particular, as an important part of the `Evrynet` ecosystem, `Evryhub` is a platform for cross-chain asset transfers, bridging digital tokens between `Evrynet` and other blockchains. `Evryhub` enriches the `Evrynet` ecosystem and also presents a unique contribution to current DeFi ecosystem.

The basic information of `Evryhub` is as follows:

Table 1.1: Basic Information of Evryhub

Item	Description
Target	Evryhub
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	November 8, 2021

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

- https://gitlab.com/Evrynet/evry_hub2/evryhub-server.git (cce81294)

And these are the commit hash values after all fixes for the issues found in the audit have been checked in:

- https://gitlab.com/Evrynet/evry_hub2/evryhub-server.git (d917d2d2)

1.2 About PeckShield

PeckShield Inc. [11] 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- 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) [9], 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 `Evryhub` 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	1
Undetermined	1
Total	5

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, 2 low-severity vulnerabilities, 1 informational recommendation, and 1 undetermined issue.

Table 2.1: Key Evryhub Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Potential Reentrancy Risk In Bridge-Bank::lock()	Time and State	Fixed
PVE-002	Low	Improved Validation Of Function Arguments	Coding Practices	Fixed
PVE-003	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated
PVE-004	Informational	Suggested Event Generation In changeOperator()	Coding Practices	Fixed
PVE-005	Undetermined	Suggested Fine-Grained Risk Control Of Transfer Volume	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 Potential Reentrancy Risk In BridgeBank::lock()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact:Medium
- Target: BridgeBank
- Category: Time and State [8]
- CWE subcategory: CWE-682 [5]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [13] exploit, and the recent `Uniswap/Lendf.Me` hack [12].

In the `Evryhub` implementation, we notice there is a routine (i.e., `lock()`) that has potential reentrancy risk. To elaborate, we show below the related code snippet of the `lock()` routine in the `BridgeBank` contract. In the function, the `BridgeToken(_token).safeTransferFrom(msg.sender, address(this), _amount)` is called (line 118 - line 122) to lock the `_token` to the `BridgeBank`. If the `_token` faithfully implements the ERC777-like standard, then the `BridgeBank::lock()` routine is vulnerable to reentrancy and this risk needs to be properly mitigated.

```
84      function lock(
85          address _recipient,
86          address _token,
87          uint256 _amount,
88          string memory _chainName
89      ) public payable availableNonce whenNotPaused {
90          string memory symbol;
91
```

```
        // ETH deposit
        if (msg.value > 0) {
            require(
                _token == address(0),
                "Ethereum deposits require the 'token' address to be the null address"
            );
            require(
                msg.value == _amount,
                "The transactions value must be equal the specified amount (in wei)"
            );
            symbol = "ETH";

            lockFunds(
            payable(msg.sender),
            _recipient,
            _token,
            symbol,
            _amount,
            _chainName
            );

        }// ERC20 deposit
        else {

            uint beforeLock = BridgeToken(_token).balanceOf(address(this));

            BridgeToken(_token).safeTransferFrom(
                msg.sender,
                address(this),
                _amount
            );

            uint afterLock = BridgeToken(_token).balanceOf(address(this));

            // Set symbol to the ERC20 token's symbol
            symbol = BridgeToken(_token).symbol();

            lockFunds(
            payable(msg.sender),
            _recipient,
            _token,
            symbol,
            afterLock - beforeLock,
            _chainName
            );
        }
    }
```

Listing 3.1: `BridgeBank::lock()`

Specifically, the ERC777 standard normalizes the ways to interact with a token contract while remaining backward compatible with ERC20. Among various features, it supports send/receive hooks to offer token holders more control over their tokens. Specifically, when `transfer()` or `transferFrom()` actions happen, the owner can be notified to make a judgment call so that she can control (or even reject) which token they send or receive by correspondingly registering `tokensToSend()` and `tokensReceived()` hooks. Consequently, any `transfer()` or `transferFrom()` of ERC777-based tokens might introduce the chance for reentrancy or hook execution for unintended purposes (e.g., mining GasTokens).

In our case, the above hook can be planted in `BridgeToken(_token).safeTransferFrom(msg.sender, address(this), _amount)` (line 118 - line 122). By doing so, we can effectively keep `beforeLock` intact (used for the calculation of actual `_token` amount transferred to the `BridgeBank` at line 134). With a lower `beforeLock`, the re-entered `BridgeBank::lock()` is able to obtain more cross-chain transfer credits. It can be repeated to exploit this vulnerability for gains, just like earlier Uniswap/imBTC hack [12].

Note that other functions `unlockFunds()` and `refunds()` can also benefit from reentrancy protection by following the known best practice of the `checks-effects-interactions` pattern.

Recommendation Add necessary reentrancy guards to prevent unwanted reentrancy risks.

Status The issue has been addressed by the following commits: `c83698f4` && `fc27f678`.

3.2 Improved Validation Of Function Arguments

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `BridgeBank`
- Category: Coding Practices [7]
- CWE subcategory: CWE-628 [3]

Description

In the `Evryhub` implementation, the `BridgeBank` contract is designed to be the main entry for interaction with users. In particular, one routine, i.e., `refund()`, is designed to refund the assets to users if the transaction of the cross-chain asset-transfer falls through. While examining the logic of the `refund()` routine, we notice the validations of the input parameters need to be enhanced.

To elaborate, we show below the related code snippet of the `refund()` function. In the function, we notice there are several validations of the input parameters, including the refund state `refundCompleted[_nonce].isRefunded` (lines 215-218), the token `refundCompleted[_nonce].tokenAddress` (lines 219-222) and the recipient `refundCompleted[_nonce].sender` (lines 223-226). However, we notice there is a lack of validation on the cross-chain transfer amount. This is reasonable under the assumption that the input `_amount` parameter is always correctly provided. However, in the unlikely situation, if the `_amount` is improperly provided (e.g., larger than the previous cross-chain transfer amount), the `BridgeBank` contract will suffer unnecessary loss. Given this, we suggest to add the validation of the input `_amount` parameter as below: `require(refundCompleted[_nonce].amount == _amount)`.

```
    function refund(
        address payable _recipient,
        address _tokenAddress,
        string memory _symbol,
        uint256 _amount,
        uint256 _nonce
    ) public onlyOperator whenNotPaused {
        require(
            refundCompleted[_nonce].isRefunded == false,
            "This refunds has been processed before"
        );
        require(
            refundCompleted[_nonce].tokenAddress == _tokenAddress,
            "This refunds has been processed before"
        );
        require(
            refundCompleted[_nonce].sender == _recipient,
            "This refunds has been processed before"
        );


        // Check if it is ETH
        if (_tokenAddress == address(0)) {
            address thisadd = address(this);
            require(
                thisadd.balance >= _amount,
                "Insufficient ethereum balance for delivery."
            );
        } else {
            require(
                BridgeToken(_tokenAddress).balanceOf(address(this)) >= _amount,
                "Insufficient ERC20 token balance for delivery."
            );
        }
        refunds(_recipient, _tokenAddress, _symbol, _amount, _nonce);
    }
```

Listing 3.2: `BridgeBank::refund()`

Recommendation Enhance the validation of the input parameters for the `refund()` routine as above-mentioned.

Status The issue has been addressed by the following commits: `c83698f4` && `fc27f678`.

3.3 Trust Issue Of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `BridgeBank`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [1]

Description

In the `Evryhub` implementation, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., manage another privilege `operator` account). In the following, we show the representative functions potentially affected by the privileged `owner`.

```
    function unlock(
        address payable _recipient,
        address tokenAddress,
        string memory _symbol,
        uint256 _amount,
        bytes32 _interchainTX
    ) public onlyOperator whenNotPaused {

        require(
            unlockCompleted[_interchainTX].isUnlocked == false,
            "Transactions has been processed before"
        );

        // Check if it is ETH
        if (tokenAddress == address(0)) {
            address thisadd = address(this);
            require(
                thisadd.balance >= _amount,
                "Insufficient ethereum balance for delivery."
            );
        } else {
            require(
                BridgeToken(tokenAddress).balanceOf(address(this)) >= _amount,
                "Insufficient ERC20 token balance for delivery."
            );
        }
        unlockFunds(_recipient, tokenAddress, _symbol, _amount, _interchainTX);
    }
```

Listing 3.3: `BridgeBank::unlock()`

```
    function emergencyWithdraw(
        address tokenAddress,
        uint256 _amount
    ) public onlyOperator whenPaused isAbleToWithdraw{
```

```

        // Check if it is ETH
        if (tokenAddress == address(0)) {
            address thisadd = address(this);
            require(
                thisadd.balance >= _amount,
                "Insufficient ethereum balance for delivery."
            );
            payable(msg.sender).transfer(_amount);
        } else {
            require(
                BridgeToken(tokenAddress).balanceOf(address(this)) >= _amount,
                "Insufficient ERC20 token balance for delivery."
            );
            BridgeToken(tokenAddress).safeTransfer(owner, _amount);
        }

    }
```

Listing 3.4: `BridgeBank::emergencyWithdraw()`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged `owner` 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 `Evryhub` design.

Recommendation Promptly transfer the privileged `owner` 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 The issue has been mitigated in the following commits by introducing `timelock` and `multi-sig` mechanism: `c83698f4` && `fc27f678`.

3.4 Suggested Event Generation In changeOperator()

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `BridgeBank`
- Category: Coding Practices [7]
- 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 BridgeBank dynamics, we notice there is a lack of emitting an event to reflect operator changes. To elaborate, we show below the related code snippet of the contract.

```
    function changeOperator(address _newOperator)
        public
        isOwner
    {
        operator = _newOperator;
    }
```

Listing 3.5: BridgeBank::changeOperator()

With that, we suggest to add a new event NewOperator whenever the new operator is changed. Also, the new operator 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 operator 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 commits: c83698f4 && fc27f678.

3.5 Suggested Fine-Grained Risk Control Of Transfer Volume

- ID: PVE-005
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: BridgeBank
- Category: Security Features [6]
- CWE subcategory: CWE-654 [4]

Description

According to the `Evryhub` design, the `BridgeBank` contract will likely accumulate a huge amount of assets with the increased popularity of cross-chain transactions. While examining the implementation of the `BridgeBank`, 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 privileged account keys are always properly managed. In the following, we take the `BridgeBank::lock()/unlock()` routines to elaborate our suggestion.

Specifically, we show below the related code snippet of the `BridgeBank` contract. According to the `Evryhub` design, when the `lock()` function is called on the source chain, the `unlock()` function on the destination chain will be called subsequently by the privileged `operator` account to transfer a certain amount of corresponding tokens to the recipient, in order to reach the cross-chain transfer purpose. In the `unlock()` function, we notice the only protection is validating the `msg.sender`. If we assume the privileged `operator` account is hijacked or leaked, all the assets locked up in the `BridgeBank` 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.

```
    /*
     * @dev: Locks received ETH/ERC20 funds.
     *
     * @param _recipient: representation of destination address.
     * @param _token: token address in origin chain (0x0 if ethereum)
     * @param _amount: value of deposit
     */
    function lock(
        address _recipient,
        address _token,
        uint256 _amount,
        string memory _chainName
    ) public payable availableNonce whenNotPaused {
        string memory symbol;

        // ETH deposit
        if (msg.value > 0) {
            require(
                _token == address(0),
                "Ethereum deposits require the 'token' address to be the null address"
            );
            require(
                msg.value == _amount,
                "The transactions value must be equal the specified amount (in wei)"
            );
            symbol = "ETH";

            lockFunds(
            payable(msg.sender),
```

```
            _recipient,
            _token,
            symbol,
            _amount,
            _chainName
            );

        }// ERC20 deposit
        else {

            uint beforeLock = BridgeToken(_token).balanceOf(address(this));

            BridgeToken(_token).safeTransferFrom(
                msg.sender,
                address(this),
                _amount
            );

            uint afterLock = BridgeToken(_token).balanceOf(address(this));

            // Set symbol to the ERC20 token's symbol
            symbol = BridgeToken(_token).symbol();

            lockFunds(
            payable(msg.sender),
            _recipient,
            _token,
            symbol,
            afterLock - beforeLock,
            _chainName
            );
        }
    }

    /*
     * @dev: Unlocks ETH and ERC20 tokens held on the contract.
     *
     * @param _recipient: recipient's is an evry address
     * @param _token: token contract address
     * @param _symbol: token symbol
     * @param _amount: wei amount or ERC20 token count
     */
    function unlock(
        address payable _recipient,
        address tokenAddress,
        string memory _symbol,
        uint256 _amount,
        bytes32 _interchainTX
    ) public onlyOperator whenNotPaused {

        require(
            unlockCompleted[_interchainTX].isUnlocked == false,
```

```
            "Transactions has been processed before"
        );

        // Check if it is ETH
        if (tokenAddress == address(0)) {
            address thisadd = address(this);
            require(
                thisadd.balance >= _amount,
                "Insufficient ethereum balance for delivery."
            );
        } else {
            require(
                BridgeToken(tokenAddress).balanceOf(address(this)) >= _amount,
                "Insufficient ERC20 token balance for delivery."
            );
        }
        unlockFunds(_recipient, tokenAddress, _symbol, _amount, _interchainTX);
    }
```

Listing 3.6: `BridgeBank::lock()&&unlock()`

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 confirmed by the team.

4 | Conclusion

In this audit, we have analyzed the `Evryhub` design and implementation. `Evrynet` is an intelligent financial services platform that provides infrastructure, which enables developers and businesses to build an unlimited number of centralized/decentralized finance (`CeDeFi`) applications. It is interoperable with many of the world's leading blockchains. In particular, as an important part of the `Evrynet` ecosystem, `Evryhub` is a platform for cross-chain asset transfer, bridging digital tokens between `Evrynet` and other blockchains. `Evryhub` enriches the `Evrynet` 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-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-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

[4] MITRE. CWE-654: Reliance on a Single Factor in a Security Decision. https://cwe.mitre.org/data/definitions/654.html.

[5] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[6] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[7] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[8] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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

[12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[13] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.