SMART CONTRACT AUDIT REPORT

for

Poly Network

Prepared By: Yiqun Chen

PeckShield

October 11, 2021

Document Properties

Client	Poly Network
Title	Smart Contract Audit Report
Target	Poly Network
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	October 11, 2021	Shulin Bie	Final Release
1.0-rc	September 27, 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 Poly 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 Poly Network

Poly Network is a cross-chain interoperability bridge (that allows a variety of chains to flexibly interact with each other and transfer arbitrary data along with carrying out cross-chain transactions). Arguably one of the largest cross-chain protocols in terms of Total Value Locked (TVL) and liquidity, it has so far supported a number of blockchains, including Ethereum, Binance Smart Chain (BSC), Polygon , Heco, Ontology, etc. Poly Network enriches the DeFi market and also presents a unique contribution to current DeFi ecosystem.

The basic information of Poly Network is as follows:

Table 1.1: Basic Information of Poly Network

Item	Description
Target	Poly Network
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	October 11, 2021

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit. Please note this audit only covers the following contract files in these two Git repositories: EthCrossChainData.sol, EthCrossChainManager.sol, UpgradableECCM.sol, EthCrossChainManagerProxy.sol

, `EthCrossChainUtils.sol`, `LockProxy.sol`, `ETH_swapper.sol`, and `SwapProxy_v2.sol`.

- https://github.com/polynetwork/eth-contracts.git (2b1cbe0)
- https://github.com/polynetwork/poly-swap.git (4ca919c)

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

- https://github.com/polynetwork/eth-contracts/pull/22/commits/c6b86da (c6b86da)

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).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

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

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

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

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.

bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Poly Network` 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	3	■ ■ ■
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 1 medium-severity vulnerability, 3 low-severity vulnerabilities, 1 informational recommendation, and 1 undetermined issue.

Table 2.1: Key Poly Network Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Improved Logic Of removeUnderlying()	Business Logic	Confirmed
PVE-002	Low	Incompatibility With Deflationary/Rebasing Tokens	Business Logic	Confirmed
PVE-003	Low	Improved Sanity Check In ECCUtils::verifySig()	Coding Practices	Fixed
PVE-004	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-005	Undetermined	Suggested Fine-Grained Risk Control Of Transfer Volume	Security Features	Confirmed
PVE-006	Low	Validation Of Whitelist Contracts And Their Methods	Business Logic	Fixed

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 Logic Of removeUnderlying()

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `SwapProxy`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the `Poly Network` implementation, the `SwapProxy` contract is designed to swap assets between blockchains. In particular, one routine, i.e., `removeUnderlying()`, is designed to remove the LP token of a specified pool to receive one underlying token of the pool and then transfer to the destination chain with cross-chain transactions. While examining the logic of the `removeUnderlying()` function, we observe an improper implementation that can be improved safely.

To elaborate, we show below the related code snippet of the `SwapProxy` contract. In the above-mentioned `removeUnderlying()` function, the following statement is executed to decide which token to receive when the pool LP token is removed: `outAssetAddress = assetPoolMap[args.toPoolId][fromChainId][args.toAssetHash]` (line 224). We notice the `outAssetAddress` is retrieved based on `args.toPoolId`, `fromChainId` and `args.toAssetHash`. However, according to the `SwapProxy` contract design, the `outAssetAddress` should be retrieved based on `args.toChainId` rather than `fromChainId` as the `swapUnderlying()` function does (line 120). Fortunately, this vulnerability can be solved by add `assetPoolMap` with the `bindPoolAssetAddress()` function though it undermines the original intention of design.

```
212     function removeUnderlying(bytes memory argsBs, bytes memory fromContractAddr, uint64
             fromChainId) onlyThis external returns (bool) {
213         SwapArgs memory args = _deserializeSwapArgs(argsBs);

215         require(fromContractAddr.length != 0, "from contract address cannot be empty");
216         require(Utils.equalStorage(swapperHashMap[fromChainId], fromContractAddr), "from
                 swapper contract address error!");
```

```
        address poolAddress = poolAddressMap[args.toPoolId];
        require(poolAddress != address(0), "pool do not exsit");
        require(Utils.equalStorage(assetHashMap[ISwap(poolAddress).lp_token()][fromChainId], args.fromAssetHash), "from Asset do not match pool token address");

        // address outAssetAddress = assetPoolMap[args.toPoolId][args.toChainId][args.toAssetHash];
        // NOT fromChainId !!!!!!!!!!
        address outAssetAddress = assetPoolMap[args.toPoolId][fromChainId][args.toAssetHash];
        require(outAssetAddress != address(0), "target asset do not exsit");

        require(args.toAddress.length != 0, "toAddress cannot be empty");

        uint outAmount = _removeInPool(poolAddress, args.amount, outAssetAddress, args.minOut);

        require(_crossChain(args.toChainId, args.toAddress, args.toAssetHash, outAmount));


        emit UnlockEvent(poolAddress, address(this), args.amount);
        emit RemoveLiquidityEvent(args.toPoolId, poolAddress, args.amount, outAssetAddress, outAmount, args.toChainId, args.toAssetHash, args.toAddress);
        emit LockEvent(outAssetAddress, address(this), args.toChainId, args.toAssetHash, args.toAddress, outAmount);

        return true;
    }
```

Listing 3.1: `SwapProxy::removeUnderlying()`

```
    function swapUnderlying(bytes memory argsBs, bytes memory fromContractAddr, uint64 fromChainId) onlyThis external returns (bool) {
        SwapArgs memory args = _deserializeSwapArgs(argsBs);

        require(fromContractAddr.length != 0, "from contract address cannot be empty");
        require(Utils.equalStorage(swapperHashMap[fromChainId], fromContractAddr), "from swapper contract address error!");

        address poolAddress = poolAddressMap[args.toPoolId];
        require(poolAddress != address(0), "pool do not exsit");

        address inAssetAddress = assetPoolMap[args.toPoolId][fromChainId][args.fromAssetHash];
        require(inAssetAddress != address(0), "inAssetHash cannot be empty");

        address outAssetAddress = assetPoolMap[args.toPoolId][args.toChainId][args.toAssetHash];
        require(outAssetAddress != address(0), "target asset do not exsit");
```

```
        require(args.toAddress.length != 0, "toAddress cannot be empty");

        require(args.toAssetHash.length != 0, "empty illegal toAssetHash");


        uint outAmount = _swapInPool(poolAddress, inAssetAddress, args.amount,
            outAssetAddress, args.minOut);

        require(_crossChain(args.toChainId, args.toAddress, args.toAssetHash, outAmount)
            );


        emit UnlockEvent(inAssetAddress, address(this), args.amount);
        emit SwapEvent(args.toPoolId, inAssetAddress, args.amount, outAssetAddress,
            outAmount, args.toChainId, args.toAssetHash, args.toAddress);
        emit LockEvent(outAssetAddress, address(this), args.toChainId, args.toAssetHash,
             args.toAddress, outAmount);

        return true;

    }
```

Listing 3.2: `SwapProxy::swapUnderlying()`

Recommendation Improve the `removeUnderlying()` routine as above-mentioned.

Status The issue has been confirmed by the team. Since the code is already deployed and the vulnerability can be solved by adjusted configuration, the team decides to resolve it in `SwapProxy V3`.

PeckShield

3.2 Incompatibility With Deflationary/Rebasing Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `LockProxy/Swapper`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the `Poly Network` implementation, the `LockProxy` contract is designed to transfer assets between blockchains. In particular, one routine, i.e., `lock()`, is designed to lock a certain amount (specified by the input `amount` parameter) of assets (specified by the input `fromAssetHash` parameter) in the source chain. Meanwhile, the equal amount of corresponding assets on the destination chain will be transferred to the recipient with the calling of `unlock()` function.

```
    function lock(address fromAssetHash, uint64 toChainId, bytes memory toAddress, uint256 amount) public payable returns (bool) {
        require(amount != 0, "amount cannot be zero!");


        require(_transferToContract(fromAssetHash, amount), "transfer asset from fromAddress to lock_proxy contract  failed!");

        bytes memory toAssetHash = assetHashMap[fromAssetHash][toChainId];
        require(toAssetHash.length != 0, "empty illegal toAssetHash");

        TxArgs memory txArgs = TxArgs({
            toAssetHash: toAssetHash,
            toAddress: toAddress,
            amount: amount
        });
        bytes memory txData = _serializeTxArgs(txArgs);

        IEthCrossChainManagerProxy eccmp = IEthCrossChainManagerProxy(managerProxyContract);
        address eccmAddr = eccmp.getEthCrossChainManager();
        IEthCrossChainManager eccm = IEthCrossChainManager(eccmAddr);

        bytes memory toProxyHash = proxyHashMap[toChainId];
        require(toProxyHash.length != 0, "empty illegal toProxyHash");
        require(eccm.crossChain(toChainId, toProxyHash, "unlock", txData), "EthCrossChainManager crossChain executed error!");

        emit LockEvent(fromAssetHash, _msgSender(), toChainId, toAssetHash, toAddress, amount);

        return true;

    }

    // /* @notice                  This function is meant to be invoked by the ETH crosschain management contract,
    // *                           then mint a certin amount of tokens to the designated address since a certain amount
    // *                           was burnt from the source chain invoker.
    // *  @param argsBs            The argument bytes recevied by the ethereum lock proxy contract, need to be deserialized.
    // *                           based on the way of serialization in the source chain proxy contract.
    // *  @param fromContractAddr  The source chain contract address
    // *  @param fromChainId       The source chain id
    // */
    function unlock(bytes memory argsBs, bytes memory fromContractAddr, uint64 fromChainId) onlyManagerContract public returns (bool) {
        TxArgs memory args = _deserializeTxArgs(argsBs);
```

```
        require(fromContractAddr.length != 0, "from proxy contract address cannot be empty");
        require(Utils.equalStorage(proxyHashMap[fromChainId], fromContractAddr), "From Proxy contract address error!");

        require(args.toAssetHash.length != 0, "toAssetHash cannot be empty");
        address toAssetHash = Utils.bytesToAddress(args.toAssetHash);

        require(args.toAddress.length != 0, "toAddress cannot be empty");
        address toAddress = Utils.bytesToAddress(args.toAddress);


        require(_transferFromContract(toAssetHash, toAddress, args.amount), "transfer asset from lock_proxy contract to toAddress failed!");

        emit UnlockEvent(toAssetHash, toAddress, args.amount);
        return true;
    }
```

Listing 3.3: `LockProxy::lock()&&unlock()`

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 `lock()/unlock()`, may introduce unexpected assets loss locked in the `LockProxy` contract because the actual amount of `ERC20` token transferred into the `LockProxy` contract is less than the cross-chain transfer 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 `LockProxy` 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 `LockProxy`. In the `Poly Network` 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. Note other routines, i.e., `Swapper::swap()/add_liquidity()/remove_liquidity()`, can also benefit from this mitigation.

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. The `Poly Network` protocol will not support deflationary (and rebasing) tokens for the time being.

3.3 Improved Sanity Check In ECCUtils::verifySig()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `ECCUtils/EthCrossChainManager`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

The `ECCUtils` library is designed to provide a set of `ECC`-related utilities for the `Poly Network` protocol. While examining the logic of these utilities, we notice there is a potential issue in the `verifySig()` function, which is widely used for signature verification.

To elaborate, we show below the related code snippet of the `verifyHeaderAndExecuteTx()` and `verifySig()` functions. In the `Poly Network` implementation, when the user initiates a cross-chain transaction on the source chain, the `verifyHeaderAndExecuteTx()` function on the destination chain will be called subsequently. After the proper `Merkle` proof and `bookkeepers` signature verification, the intended cross-chain transaction on the destination chain will be executed. In the `verifyHeaderAndExecuteTx()` function, the `verifySig()` function is called (lines 173 and 176) to ensure the transaction has been signed by at least $\frac{2}{3}$ `bookkeepers`.

While examining the logic of the `verifySig()` function, we notice it internally calls `ecrecover()` (line 126), which is one pre-compiled contract. The purpose here is to compute the public key corresponding to the private key that was used to create an `ECDSA` signature. It returns the recovered address associated with the public key or returns zero on error. With that, if we assume the `ConKeepersPkBytes` storage variable in the `EthCrossChainData` contract that stores the `bookkeepers` public keys includes `address(0)`. The signature verification in the `verifyHeaderAndExecuteTx()` function may be bypassed. We suggest to improve the `verifySig()` function by further enforcing the following sanity check: `if (signers[j] == address(0))return false`.

```
160     function verifyHeaderAndExecuteTx(bytes memory proof, bytes memory rawHeader, bytes
            memory headerProof, bytes memory curRawHeader,bytes memory headerSig)
            whenNotPaused public returns (bool){
161         ECCUtils.Header memory header = ECCUtils.deserializeHeader(rawHeader);
162         // Load ehereum cross chain data contract
163         IEthCrossChainData eccd = IEthCrossChainData(EthCrossChainDataAddress);
164
```

```
        // Get stored consensus public key bytes of current poly chain epoch and
            deserialize Poly chain consensus public key bytes to address[]
        address[] memory polyChainBKs = ECCUtils.deserializeKeepers(eccd.
            getCurEpochConPubKeyBytes());

        uint256 curEpochStartHeight = eccd.getCurEpochStartHeight();

        uint n = polyChainBKs.length;
        if (header.height >= curEpochStartHeight) {
            // It's enough to verify rawHeader signature
            require(ECCUtils.verifySig(rawHeader, headerSig, polyChainBKs, n - ( n - 1)
                / 3), "Verify poly chain header signature failed!");
        } else {
            // We need to verify the signature of curHeader
            require(ECCUtils.verifySig(curRawHeader, headerSig, polyChainBKs, n - ( n -
                1) / 3), "Verify poly chain current epoch header signature failed!");

            // Then use curHeader.StateRoot and headerProof to verify rawHeader.
                CrossStateRoot
            ECCUtils.Header memory curHeader = ECCUtils.deserializeHeader(curRawHeader);
            bytes memory proveValue = ECCUtils.merkleProve(headerProof, curHeader.
                blockRoot);
            require(ECCUtils.getHeaderHash(rawHeader) == Utils.bytesToBytes32(proveValue
                ), "verify header proof failed!");
        }

        // Through rawHeader.CrossStatesRoot, the toMerkleValue or cross chain msg can
            be verified and parsed from proof
        bytes memory toMerkleValueBs = ECCUtils.merkleProve(proof, header.
            crossStatesRoot);

        // Parse the toMerkleValue struct and make sure the tx has not been processed,
            then mark this tx as processed
        ECCUtils.ToMerkleValue memory toMerkleValue = ECCUtils.deserializeMerkleValue(
            toMerkleValueBs);
        require(!eccd.checkIfFromChainTxExist(toMerkleValue.fromChainID, Utils.
            bytesToBytes32(toMerkleValue.txHash)), "the transaction has been executed!")
            ;
        require(eccd.markFromChainTxExist(toMerkleValue.fromChainID, Utils.
            bytesToBytes32(toMerkleValue.txHash)), "Save crosschain tx exist failed!");

        // Ethereum ChainId is 2, we need to check the transaction is for Ethereum
            network
        require(toMerkleValue.makeTxParam.toChainId == chainId, "This Tx is not aiming
            at this network!");

        // Obtain the targeting contract, so that Ethereum cross chain manager contract
            can trigger the executation of cross chain tx on Ethereum side
        address toContract = Utils.bytesToAddress(toMerkleValue.makeTxParam.toContract);

        // only invoke PreWhiteListed Contract and method For Now
        require(whiteListToContract[toContract],"Invalid to contract");
```

```
        require(whiteListMethod[toMerkleValue.makeTxParam.method],"Invalid method");

        //TODO: check this part to make sure we commit the next line when doing local
            net UT test
        require(_executeCrossChainTx(toContract, toMerkleValue.makeTxParam.method,
            toMerkleValue.makeTxParam.args, toMerkleValue.makeTxParam.fromContract,
            toMerkleValue.fromChainID), "Execute CrossChain Tx failed!");

        // Fire the cross chain event denoting the executation of cross chain tx is
            successful,
        // and this tx is coming from other public chains to current Ethereum network
        emit VerifyHeaderAndExecuteTxEvent(toMerkleValue.fromChainID, toMerkleValue.
            makeTxParam.toContract, toMerkleValue.txHash, toMerkleValue.makeTxParam.
            txHash);

        return true;
    }
```

Listing 3.4: EthCrossChainManager::verifyHeaderAndExecuteTx()

```
    function verifySig(bytes memory _rawHeader, bytes memory _sigList, address[] memory
        _keepers, uint _m) internal pure returns (bool){
        bytes32 hash = getHeaderHash(_rawHeader);

        uint sigCount = _sigList.length.div(POLYCHAIN_SIGNATURE_LEN);
        address[] memory signers = new address[](sigCount);
        bytes32 r;
        bytes32 s;
        uint8 v;
        for(uint j = 0; j  < sigCount; j++){
            r = Utils.bytesToBytes32(Utils.slice(_sigList, j*POLYCHAIN_SIGNATURE_LEN,
                32));
            s =  Utils.bytesToBytes32(Utils.slice(_sigList, j*POLYCHAIN_SIGNATURE_LEN +
                32, 32));
            v =  uint8(_sigList[j*POLYCHAIN_SIGNATURE_LEN + 64]) + 27;
            signers[j] =  ecrecover(sha256(abi.encodePacked(hash)), v, r, s);
        }
        return Utils.containMAddresses(_keepers, signers, _m);
    }
```

Listing 3.5: ECCUtils::verifySig()

Recommendation Properly handle the situation when the ecrecover() routine returns zero on error.

Status The issue has been addressed in the following commit: c6b86da.

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

```
    function update(address[] memory tokens, address newContract) onlyOwner public {
        for (uint i=0;i<tokens.length;i++) {
            require(_transferERC20FromContract(tokens[i], newContract, IERC20(tokens[i])
                .balanceOf(address(this))),
            "transfer asset to newContract failed!");
        }
    }
```

Listing 3.6: `SwapProxy::update()`

```
    function unlock(bytes memory argsBs, bytes memory fromContractAddr, uint64
        fromChainId) onlyManagerContract public returns (bool) {
        TxArgs memory args = _deserializeTxArgs(argsBs);

        require(fromContractAddr.length != 0, "from proxy contract address cannot be
            empty");
        require(Utils.equalStorage(proxyHashMap[fromChainId], fromContractAddr), "From
            Proxy contract address error!");

        require(args.toAssetHash.length != 0, "toAssetHash cannot be empty");
        address toAssetHash = Utils.bytesToAddress(args.toAssetHash);

        require(args.toAddress.length != 0, "toAddress cannot be empty");
        address toAddress = Utils.bytesToAddress(args.toAddress);


        require(_transferFromContract(toAssetHash, toAddress, args.amount), "transfer
            asset from lock_proxy contract to toAddress failed!");

        emit UnlockEvent(toAssetHash, toAddress, args.amount);
        return true;
    }
```

Listing 3.7: `LockProxy::unlock()`

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 `Poly Network` 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 This issue has been confirmed by the team. The team intends to introduce `PoS` consensus mechanism in `Poly Network V2.0`.

3.5 Suggested Fine-Grained Risk Control Of Transfer Volume

- ID: PVE-005
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `LockProxy`
- Category: Security Features [5]
- CWE subcategory: CWE-654 [3]

Description

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

Specifically, we show below the related code snippet of the `LockProxy` contract. According to the `Poly Network` design, when the `lock()` function is called on the source chain, the `EthCrossChainManager ::verifyHeaderAndExecuteTx()` function on the destination chain will be called subsequently. After the proper `Merkle` proof and `bookkeepers` signature verification, the `unlock()` function will be called finally to transfer a certain amount of assets to the recipient, in order to reach the cross-chain transfer purpose. Considering the unlikely situation where the `bookkeeper keys` may be hijacked or leaked, all the assets locked up in the `LockProxy` 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.

```
function lock(address fromAssetHash, uint64 toChainId, bytes memory toAddress, uint256 amount) public payable returns (bool) {
    require(amount != 0, "amount cannot be zero!");


    require(_transferToContract(fromAssetHash, amount), "transfer asset from fromAddress to lock_proxy contract  failed!");

    bytes memory toAssetHash = assetHashMap[fromAssetHash][toChainId];
    require(toAssetHash.length != 0, "empty illegal toAssetHash");

    TxArgs memory txArgs = TxArgs({
        toAssetHash: toAssetHash,
        toAddress: toAddress,
        amount: amount
    });
    bytes memory txData = _serializeTxArgs(txArgs);

    IEthCrossChainManagerProxy eccmp = IEthCrossChainManagerProxy(managerProxyContract);
    address eccmAddr = eccmp.getEthCrossChainManager();
    IEthCrossChainManager eccm = IEthCrossChainManager(eccmAddr);

    bytes memory toProxyHash = proxyHashMap[toChainId];
    require(toProxyHash.length != 0, "empty illegal toProxyHash");
    require(eccm.crossChain(toChainId, toProxyHash, "unlock", txData), "EthCrossChainManager crossChain executed error!");

    emit LockEvent(fromAssetHash, _msgSender(), toChainId, toAssetHash, toAddress, amount);

    return true;

}

// /* @notice                  This function is meant to be invoked by the ETH crosschain management contract,
// *                           then mint a certin amount of tokens to the designated address since a certain amount
// *                           was burnt from the source chain invoker.
// *  @param argsBs            The argument bytes recevied by the ethereum lock proxy contract, need to be deserialized.
// *                           based on the way of serialization in the source chain proxy contract.
// *  @param fromContractAddr  The source chain contract address
// *  @param fromChainId       The source chain id
// */
function unlock(bytes memory argsBs, bytes memory fromContractAddr, uint64 fromChainId) onlyManagerContract public returns (bool) {
    TxArgs memory args = _deserializeTxArgs(argsBs);

    require(fromContractAddr.length != 0, "from proxy contract address cannot be empty
```

```
            ");
        require(Utils.equalStorage(proxyHashMap[fromChainId], fromContractAddr), "From
            Proxy contract address error!");

        require(args.toAssetHash.length != 0, "toAssetHash cannot be empty");
        address toAssetHash = Utils.bytesToAddress(args.toAssetHash);

        require(args.toAddress.length != 0, "toAddress cannot be empty");
        address toAddress = Utils.bytesToAddress(args.toAddress);


        require(_transferFromContract(toAssetHash, toAddress, args.amount), "transfer
            asset from lock_proxy contract to toAddress failed!");

        emit UnlockEvent(toAssetHash, toAddress, args.amount);
        return true;
    }
```

Listing 3.8: `LockProxy::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. Considering the code is alive on the mainnet, the team intends to leave it as is.

3.6 Validation Of Whitelist Contracts And Their Methods

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `EthCrossChainManager`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the `Poly Network` implementation, when the user initiates a cross-chain transaction on the source chain, the `verifyHeaderAndExecuteTx()` function on the destination chain will be called subsequently. In the `verifyHeaderAndExecuteTx()` function, we notice a much-needed whitelist mechanism, which in essence defines the list of administrator-approved contracts (specified by the `whiteListToContract` map) and methods (specified by the `whiteListMethod` map) that can be applied to validate the `toContract` and `method` to thwart any unwanted manipulation. While examining the whitelist mechanism, we notice the current whitelist mechanism can be improved.

To elaborate, we show below the related code snippet of the `verifyHeaderAndExecuteTx()` function. Specially, we notice two requirements, i.e., `require(whiteListToContract[toContract],"Invalid to contract")` (line 199) and `require(whiteListMethod[toMerkleValue.makeTxParam.method],"Invalid method")` (line 200), are used to validate whether the `toContract` and `method` are administrator-approved. However, we notice there is a corner case where the validation can be bypassed. If we assume `contractA` and `contractB` are both in the `whiteListToContract`, the `functionC()` of the `contractA` is approved by administrator and the `contractB` has the same `functionC()`, the calling of the `contractB::functionC()` will bypass the whitelist mechanism though the `functionC()` of the `contractB` has not been approved by administrator. Given this, we suggest to improve the whitelist mechanism by correlating administrator-approved contracts and method as below: `mapping(address => mapping(bytes => bool)) whiteListContractMethodMap`.

```
160     function verifyHeaderAndExecuteTx(bytes memory proof, bytes memory rawHeader, bytes
            memory headerProof, bytes memory curRawHeader,bytes memory headerSig)
            whenNotPaused public returns (bool){
161         ECCUtils.Header memory header = ECCUtils.deserializeHeader(rawHeader);
162         // Load ehereum cross chain data contract
163         IEthCrossChainData eccd = IEthCrossChainData(EthCrossChainDataAddress);

165         // Get stored consensus public key bytes of current poly chain epoch and
                deserialize Poly chain consensus public key bytes to address[]
166         address[] memory polyChainBKs = ECCUtils.deserializeKeepers(eccd.
                getCurEpochConPubKeyBytes());

168         uint256 curEpochStartHeight = eccd.getCurEpochStartHeight();

170         uint n = polyChainBKs.length;
171         if (header.height >= curEpochStartHeight) {
172             // It's enough to verify rawHeader signature
173             require(ECCUtils.verifySig(rawHeader, headerSig, polyChainBKs, n - ( n - 1)
                    / 3), "Verify poly chain header signature failed!");
174         } else {
175             // We need to verify the signature of curHeader
176             require(ECCUtils.verifySig(curRawHeader, headerSig, polyChainBKs, n - ( n -
                    1) / 3), "Verify poly chain current epoch header signature failed!");

178             // Then use curHeader.StateRoot and headerProof to verify rawHeader.
                    CrossStateRoot
179             ECCUtils.Header memory curHeader = ECCUtils.deserializeHeader(curRawHeader);
180             bytes memory proveValue = ECCUtils.merkleProve(headerProof, curHeader.
                    blockRoot);
181             require(ECCUtils.getHeaderHash(rawHeader) == Utils.bytesToBytes32(proveValue
                    ), "verify header proof failed!");
182         }

184         // Through rawHeader.CrossStatesRoot, the toMerkleValue or cross chain msg can
                be verified and parsed from proof
185         bytes memory toMerkleValueBs = ECCUtils.merkleProve(proof, header.
                crossStatesRoot);
```

```
        // Parse the toMerkleValue struct and make sure the tx has not been processed, then mark this tx as processed
        ECCUtils.ToMerkleValue memory toMerkleValue = ECCUtils.deserializeMerkleValue(toMerkleValueBs);
        require(!eccd.checkIfFromChainTxExist(toMerkleValue.fromChainID, Utils.bytesToBytes32(toMerkleValue.txHash)), "the transaction has been executed!");
        require(eccd.markFromChainTxExist(toMerkleValue.fromChainID, Utils.bytesToBytes32(toMerkleValue.txHash)), "Save crosschain tx exist failed!");

        // Ethereum ChainId is 2, we need to check the transaction is for Ethereum network
        require(toMerkleValue.makeTxParam.toChainId == chainId, "This Tx is not aiming at this network!");

        // Obtain the targeting contract, so that Ethereum cross chain manager contract can trigger the executation of cross chain tx on Ethereum side
        address toContract = Utils.bytesToAddress(toMerkleValue.makeTxParam.toContract);

        // only invoke PreWhiteListed Contract and method For Now
        require(whiteListToContract[toContract],"Invalid to contract");
        require(whiteListMethod[toMerkleValue.makeTxParam.method],"Invalid method");

        //TODO: check this part to make sure we commit the next line when doing local net UT test
        require(_executeCrossChainTx(toContract, toMerkleValue.makeTxParam.method, toMerkleValue.makeTxParam.args, toMerkleValue.makeTxParam.fromContract, toMerkleValue.fromChainID), "Execute CrossChain Tx failed!");

        // Fire the cross chain event denoting the executation of cross chain tx is successful,
        // and this tx is coming from other public chains to current Ethereum network
        emit VerifyHeaderAndExecuteTxEvent(toMerkleValue.fromChainID, toMerkleValue.makeTxParam.toContract, toMerkleValue.txHash, toMerkleValue.makeTxParam.txHash);

        return true;
    }
```

Listing 3.9: `EthCrossChainManager::verifyHeaderAndExecuteTx()`

Recommendation Improve the whitelist mechanism by binding the approved method with the approved contract together.

Status The issue has been addressed in the following commit: `c6b86da`.

4 Conclusion

In this audit, we have analyzed the `Poly Network` design and implementation. `Poly Network` is a cross-chain interoperability bridge (that allows a variety of chains to flexibly interact with each other and transfer arbitrary data along with carrying out cross-chain transactions). `Poly Network` enriches the DeFi market 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.

PeckShield

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-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.

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

PeckShield