SECURITY PATCH REVIEW

for

PolyNetwork Eth-Contracts (PR-12)

Prepared By: Yiqun Chen

PeckShield

August 15, 2021

Document Properties

Client	PolyNetwork
Title	PR-12 Security Patch Review
Target	eth-contracts
Version	1.0
Author	Xuxian Jiang
Auditors	Shulin Bie, Jing Wang, Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 15, 2021	Xuxian Jiang	Final Release
1.0-rc	August 14, 2021	Xuxian Jiang	Release Candidate #1

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

Security Patch Review

Given the opportunity to perform an informal security review of the Pull Request 12 (abbreviated as PR-12) of the PolyNetwork's eth-contracts repository, we summarize in the report our assessment to evaluate the design goal, expose potential security issues, and examine semantic inconsistencies, if any, in the given smart contract implementation. Our analysis shows that the given PR-12 implementation provides the much-needed whitelist feature (in EthCrossChainManager) that prevents the unwanted invocation of arbitrary destination contracts and their methods, hence fixing the critical vulnerability that was exploited in the PolyNetwork incident on August 10, 2021.

Review Target: https://github.com/polynetwork/eth-contracts/pull/12/files (PR-12)

Review Period: August 13, 2021 - August 15, 2021

Issue Description: PolyNetwork 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. To facilitate the implementation, the protocol has designed a number of cooperative components (each with its own roles and responsibilities), such as Relay Chain, Off-Chain Relayer, Keeper, as well as various smart contracts deployed on supported blockchains. In the following, we mainly focus on the smart contracts deployed on Ethereum as reflected in our review target.

The deployed version has an issue in blindly trusting the messages mined in the Relay Chain and failing to thoroughly and properly sanitize the deserialized message content (in EthCrossChainManager::verifyHeaderAndExecuteTx()) before carrying out cross-chain transactions. Undoubtedly, the incident is a well-executed exploitation of the above issue that originates from Ontology and propagates to BSC, Ethereum, Polygon, and Heco [1].

To elaborate, we show below the full implementation of the related verifyHeaderAndExecuteTx()

[1]Note the attempt on Heco is not successful as the related relayer does not behave exactly the same as others. The exact reason is out of scope of this audit, but is also a "TASTY FOOD FOR THE RESEARCHERS."

function. This function is designed to verify a given relay chain message (with the related header and the associated Merkle proof) and execute the intended cross-chain transaction on the destination chain, e.g., BSC, Ethereum, Polygon, etc.

```
127     function verifyHeaderAndExecuteTx(bytes memory proof, bytes memory rawHeader, bytes
            memory headerProof, bytes memory curRawHeader,bytes memory headerSig)
            whenNotPaused public returns (bool){
128         ECCUtils.Header memory header = ECCUtils.deserializeHeader(rawHeader);
129         // Load ehereum cross chain data contract
130         IEthCrossChainData eccd = IEthCrossChainData(EthCrossChainDataAddress);
131
132         // Get stored consensus public key bytes of current poly chain epoch and
                deserialize Poly chain consensus public key bytes to address[]
133         address[] memory polyChainBKs = ECCUtils.deserializeKeepers(eccd.
                getCurEpochConPubKeyBytes());
134
135         uint256 curEpochStartHeight = eccd.getCurEpochStartHeight();
136
137         uint n = polyChainBKs.length;
138         if (header.height >= curEpochStartHeight) {
139             // It's enough to verify rawHeader signature
140             require(ECCUtils.verifySig(rawHeader, headerSig, polyChainBKs, n - ( n - 1)
                    / 3), "Verify poly chain header signature failed!");
141         } else {
142             // We need to verify the signature of curHeader
143             require(ECCUtils.verifySig(curRawHeader, headerSig, polyChainBKs, n - ( n -
                    1) / 3), "Verify poly chain current epoch header signature failed!");
144
145             // Then use curHeader.StateRoot and headerProof to verify rawHeader.
                    CrossStateRoot
146             ECCUtils.Header memory curHeader = ECCUtils.deserializeHeader(curRawHeader);
147             bytes memory proveValue = ECCUtils.merkleProve(headerProof, curHeader.
                    blockRoot);
148             require(ECCUtils.getHeaderHash(rawHeader) == Utils.bytesToBytes32(proveValue
                    ), "verify header proof failed!");
149         }
150
151         // Through rawHeader.CrossStatesRoot, the toMerkleValue or cross chain msg can
                be verified and parsed from proof
152         bytes memory toMerkleValueBs = ECCUtils.merkleProve(proof, header.
                crossStatesRoot);
153
154         // Parse the toMerkleValue struct and make sure the tx has not been processed,
                then mark this tx as processed
155         ECCUtils.ToMerkleValue memory toMerkleValue = ECCUtils.deserializeMerkleValue(
                toMerkleValueBs);
156         require(!eccd.checkIfFromChainTxExist(toMerkleValue.fromChainID, Utils.
                bytesToBytes32(toMerkleValue.txHash)), "the transaction has been executed!")
                ;
157         require(eccd.markFromChainTxExist(toMerkleValue.fromChainID, Utils.
                bytesToBytes32(toMerkleValue.txHash)), "Save crosschain tx exist failed!");
158
```

```
159         // Ethereum ChainId is 2, we need to check the transaction is for Ethereum
                network
160         require(toMerkleValue.makeTxParam.toChainId == uint64(2), "This Tx is not aiming
                 at Ethereum network!");
161
162         // Obtain the targeting contract, so that Ethereum cross chain manager contract
                can trigger the execution of cross chain tx on Ethereum side
163         address toContract = Utils.bytesToAddress(toMerkleValue.makeTxParam.toContract);
164
165         //TODO: check this part to make sure we commit the next line when doing local
                net UT test
166         require(_executeCrossChainTx(toContract, toMerkleValue.makeTxParam.method,
                toMerkleValue.makeTxParam.args, toMerkleValue.makeTxParam.fromContract,
                toMerkleValue.fromChainID), "Execute CrossChain Tx failed!");
167
168         // Fire the cross chain event denoting the executation of cross chain tx is
                successful,
169         // and this tx is coming from other public chains to current Ethereum network
170         emit VerifyHeaderAndExecuteTxEvent(toMerkleValue.fromChainID, toMerkleValue.
                makeTxParam.toContract, toMerkleValue.txHash, toMerkleValue.makeTxParam.
                txHash);
171
172         return true;
173     }
```

Listing 1: `EthCrossChainManager::verifyHeaderAndExecuteTx()`

```
26     struct TxParam {
27         bytes txHash; //  source chain txhash
28         bytes crossChainId;
29         bytes fromContract;
30         uint64 toChainId;
31         bytes toContract;
32         bytes method;
33         bytes args;
34     }
```

Listing 2: `EthCrossChainUtils::ECCUtils()`

The issue stems from the insufficient validation on the deserialized `toMerkleValue` struct (line 155) after the proper `Merkle` proof and signature verification. However, its `makeTxParam` member field may contain well-crafted content: for example, the three struct sub-fields `toContract`, `method`, and `args` could be together misused to invoke a privileged call (`putCurEpochConPubKeyBytes()`) on `EthCrossChainData` to change the effective keepers. Notice that this privileged call is declared as an `onlyOwner` operation. However, it is now misused, in a manner similar to the traditional `return-to-libc (ret2libc)` attack, to take over the current keepers! To recap, we show below how a malicious cross-chain transaction may deviate from a normal one.

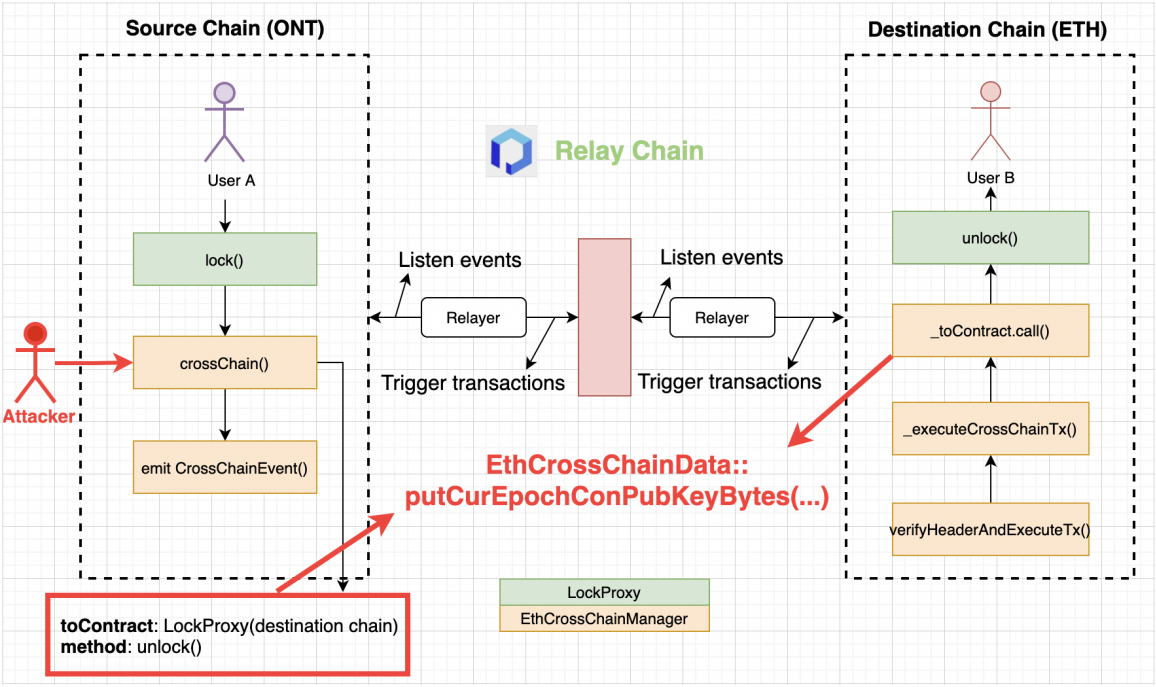

Figure 1: The Cross-Chain Transaction Comparison in PolyNetwork: Normal vs. Exploited

Issue Fixup: The PR-12 is proposed to address the above issue by implementing a much-needed whitelist feature. This whitelist feature in essence defines the list of administrator-approved contracts as well as the associated methods that are then applied to validate the above member fields, especially toContract and method, to thwart any manipulation. For extra precaution, we also make the suggestion to define a whitelist that may be allowed to call the crossChain() function to initiate a cross-chain transaction. After discussion, the team takes the suggestion and includes it a part of this PR-12.

```
160     {
161         ...
162         // Obtain the targeting contract, so that Ethereum cross chain manager contract
                can trigger the execution of cross chain tx on Ethereum side
163         address toContract = Utils.bytesToAddress(toMerkleValue.makeTxParam.toContract);
164
165 +       // only invoke PreWhiteListed Contract and method For Now
166 +       require(whiteListToContract[toContract],"Invalid to contract");
167 +       require(whiteListMethod[toMerkleValue.makeTxParam.method],"Invalid method");
168
169         //TODO: check this part to make sure we commit the next line when doing local
                net UT test
170         require(_executeCrossChainTx(toContract, toMerkleValue.makeTxParam.method,
                toMerkleValue.makeTxParam.args, toMerkleValue.makeTxParam.fromContract,
                toMerkleValue.fromChainID), "Execute CrossChain Tx failed!");
171
```

```
172         // Fire the cross chain event denoting the executation of cross chain tx is
                successful,
173         // and this tx is coming from other public chains to current Ethereum network
174         emit VerifyHeaderAndExecuteTxEvent(toMerkleValue.fromChainID, toMerkleValue.
                makeTxParam.toContract, toMerkleValue.txHash, toMerkleValue.makeTxParam.
                txHash);
175
176         return true;
177     }
```

Listing 3: The Revised `EthCrossChainManager::verifyHeaderAndExecuteTx()`

To conclude, the proposed `PR-12` achieves the intended goal by fixing the loophole in the original implementation. Once merged, it is ready to be deployed to upgrade (and fix) the deployed version.

Disclaimer

This is an informal security review, not a full security audit, and it does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. Furthermore, we always recommend proceeding with several independent full audits and a public bug bounty program to ensure the security of smart contract(s). Lastly, this security review report should not be used as investment advice.