Security Audit Report for Teleport

Date: March 30, 2022

Version: 1.0

Contact: contact@blocksecteam.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2022.03.30 16:37:38 +08'00'

Contents

Report Manifest

Item	Description
Client	Teleport-network
Target	Teleport

Version History

Version	Date	Description
1.0	March 30, 2022	First Release

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The repositories that are audited in this report include the following ones.

Repo Name	Github URL
Teleport	https://github.com/teleport-network/teleport
Xibc-contracts	https://github.com/teleport-network/xibc-contracts

The commit SHA values during the audit are shown in the following.

Project	Commit SHA
Version 1	
Teleport	949413a5ef30de94dd955bdac4fc13eb4aee1d4d
Xibc-contracts	28546637803df3de2b1ee6506cb482131d7308d5
Version 2	
Teleport	056abf50abc2e16ff82df03af437506d0b691ad2
Xibc-contracts	dd9bdb31c0e77990b281960a7dc97afe506054b5

Note that, we did **NOT** audit all the modules in the Teleport repository. Specifically, the modules covered in this audit include:

- /x/aggregate
- /x/xibc

Furthermore, the two modules have a few Cosmos modules or repositories as dependencies, which are **NOT** covered in the audit. Specifically, the modules that are not covered in this audit include:

- bank: github.com/cosmos/cosmos-sdk/x/genutil
- ethermint: github.com/tharsis/ethermint
- go-ethereum: github.com/ethereum/go-ethereum
- staking: github.com/cosmos/cosmos-sdk/x/staking
- ics23: github.com/confio/ics23/go
- tendermint: github.com/tendermint/tendermint
- codec: github.com/cosmos/cosmos-sdk/codec

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics

of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Access control

- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [1] and Common Weakness Enumeration [2]. Accordingly, the severity measured in this report are classified into four categories: **High**, **Medium**, **Low** and **Undetermined**.

Furthermore, the status of a discovered issue will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The issue has been received by the client, but not confirmed yet.
- **Confirmed** The issue has been recognized by the client, but not fixed yet.
- **Fixed** The issue has been confirmed and fixed by the client.

[1] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[2] https://cwe.mitre.org/

Chapter 2 Audit Process

This project is a cross-chain solution, which contains a large number lines of code. In particular, it includes $7,612$ lines of Solidity code and $16,195$ lines in Golang code (after removing the automatically generated one). To make the audit process clear, we will elaborate on our audit process in this chapter.

This chapter is organized as following: Section 2.1 takes a cross-chain transaction as example to introduce how Teleport works. After that, Section 2.2, Section 2.3, Section 2.4, and Section 2.5 analyzes the attack surfaces from the interfaces for accounts in supported chains, accounts in Teleport chain, relayers, and the Teleport chain itself, respectively.

2.1 A cross-chain transaction

As depicted in the Figure 2.1, suppose Bob attempts to transfer *x* ETH to his BSC account. The ① to ⓑ indicates the order. The cross-chain transaction works as following:

1. Bob invokes the xibc-contracts that are deployed on Ethereum to lock *x* ETH and indicate cross-chain transfer metadata. (①)
2. The xibc-contracts emits a specific event [1] that contains the cross-chain packet1. (②)
3. A relayer monitors the cross-chain event, and then extracts and forwards the cross-chain packet1 to the *xibc* module of Teleport.
4. The *xibc* module of Teleport then verifies if the cross-chain packet1 is emitted by the xibc-contracts. If so, it also emits a specific event that contains the cross-chain packet2. Note that the cross-chain transfer metadata stored in the packet1 and packet2 is the same, since the Teleport acts as a relay chain in this cross-chain transaction. (③ and ④)
5. Another one relayer monitors the cross-chain event from Teleport, and then sends the cross-chain packet2 to the xibc-contracts that deployed on the BSC.
6. The xibc-contracts verifies if the cross-chain packet2 is emitted by the *xibc* module of Teleport, if so, it mints *x* XIBC-ETH to Bob's BSC account that stored in the cross-chain transfer metadata, and it then emits an event that contains the ack-packet1, which includes information about if the transaction is executed successfully. (⑤, ⑥, and ⑦)
7. The forwarding and verification for ack-packets is similar with above processes. (⑧, ⑨, and ⓐ)
8. If the ack-packet2 says that the cross-chain transfer is not successfully executed at the BSC chain, the xibc-contracts in Ethereum will unlock the *x* ETH, which guarantees the atomicity of the cross-chain transaction. (ⓑ)

We believe the most significant part of the whole process is that how a blockchain verifies if a smart contract on another blockchain emits a specific event (③, ⑤, and ⑧).

Specifically, Teleport chain acts not only as an EVM compatible chain but also as a relay chain. It not only receives and executes cross-chain transactions targeting itself but also forwards cross-chain transactions that do not target itself. The benefit of this design is that supported chains only need to verify cross-chain events from the relay chain (Teleport), which reduces the size of smart contracts. On the contrary, Teleport chain needs to have the ability to verify cross-chain events on all supported chains.

[1] https://docs.soliditylang.org/en/v0.8.12/abi-spec.html#events

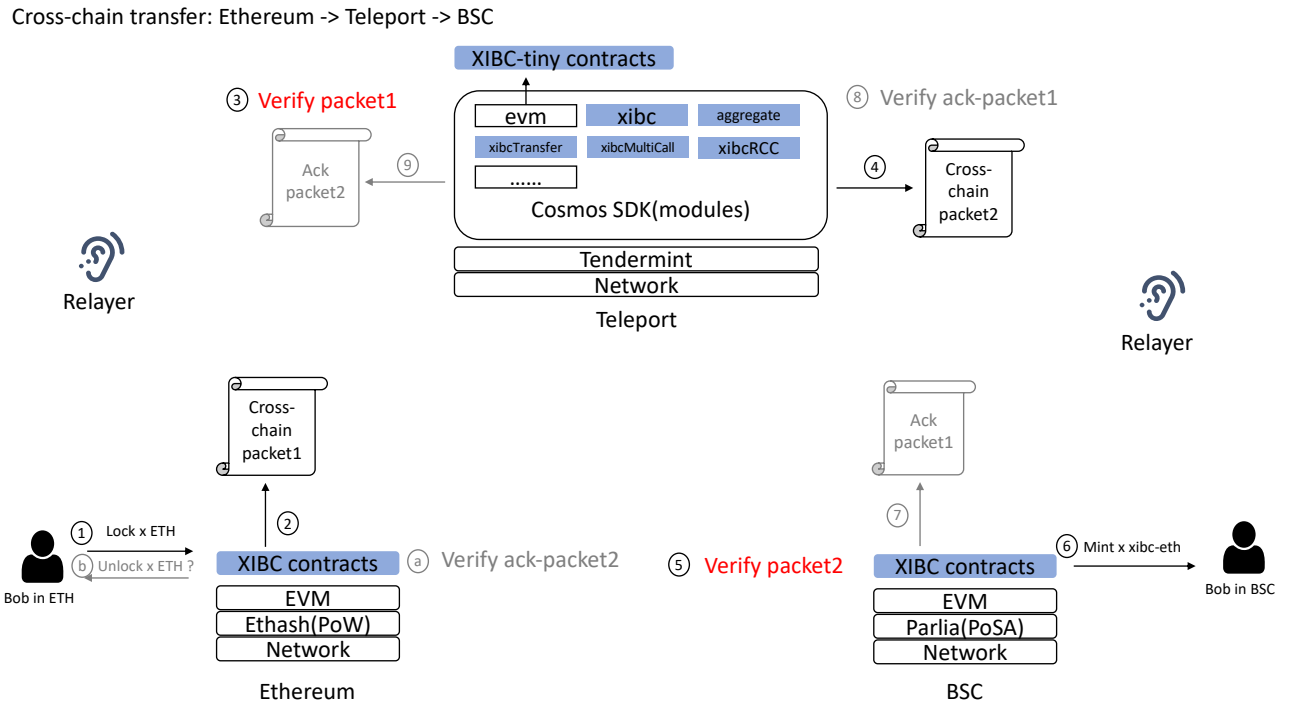

Figure 2.1: A cross-chain transaction in the Teleport project that transfers *x* ETH from an Ethereum account to a Binance Smart Chain(BSC) account. The black color represents the first two stages: the launch and execution of the cross-chain transaction, and the gray color represents the third stage: the acknowledgement for the cross-chain transaction. Relayers are a group of people (or programs) which monitor cross-chain events and send cross-chain metadata between chains.

Besides, the *xibc* module of the Teleport chain contains three light clients: eth, bsc, and tendermint, which represents an Ethereum light client, a Binance Smart Chain (BSC) light client, and a tendermint client that can verify headers from Cosmos chains. The xibc-contract that deployed on supported chains contains a tendermint light client. Relayers monitor and forward newly added headers from all chains at any time, then these clients verify headers and save the state that necessary to verify next headers and cross-chain events.

According to the cross-chain transaction in Figure 2.1, we can get the whole architecture of Teleport. In the following, we enumerate and analyze all potential attack surfaces by looking into the interfaces that can be accessed by different roles (or accounts). We list all interfaces on the table 2.1.

2.2 Interfaces for accounts in supported chains

```
255    function sendTransferBase(
256        TransferDataTypes.BaseTransferData calldata transferData
257    ) external payable override {
258        string memory sourceChain = clientManager.getChainName();
259        require(
260            !sourceChain.equals(transferData.destChain),
261            "sourceChain can't be equal to destChain"
262        );
263
264        require(msg.value > 0, "value must be greater than 0");
```

Contract (Module) Name	File Location	Function Name
Interfaces for accounts in EVM compatible chains		
Transfer	xibc-contracts/evm/contracts/apps/transfer/Transfer.sol	sendTransferBase
		sendTransferERC20
RCC	xibc-contracts/evm/contracts/apps/rcc/RCC.sol	sendRemoteContractCall
MultiCall	xibc-contracts/evm/contracts/apps/multicall/MultiCall.sol	multiCall
Interfaces for accounts in Teleport chain		
Transfer	xibc-contracts/teleport/contracts/apps/transfer/Transfer.sol	sendTransferBase
		sendTransferERC20
RCC	xibc-contracts/teleport/contracts/apps/rcc/RCC.sol	sendRemoteContractCall
MultiCall	xibc-contracts/teleport/contracts/apps/multicall/MultiCall.sol	multiCall
Interfaces for relayers		
Packet	xibc-contracts/evm/contracts/core/packet/Packet.sol	recvPacket*
		acknowledgePacket*
ClientManager	xibc-contracts/evm/contracts/core/client/ClientManager.sol	updateClient
xibc	teleport/x/xibc/keeper/msg_server.go	RecvPacket*
		Acknowledgement*
		UpdateClient
Interfaces for the Teleport project		
ClientManager	xibc-contracts/evm/contracts/core/client/ClientManager.sol	createClient
		upgradeClient
		registerRelayer
xibc	teleport/x/xibc/core/client/keeper/client.go	CreateClient
		UpgradeClient
xibc	teleport/x/xibc/core/client/keeper/relayer.go	RegisterRelayer
aggregate	teleport/x/aggregate/keeper/proposals.go	RegisterCoin
		RegisterERC20
		ToggleRelay
		UpdateTokenPairERC20
		RegisterERC20Trace

*: These four functions do not check if the caller is a registered relayer. Therefore, these functions can be called by anyone.

Table 2.1: All interfaces of Teleport

```
265
266         outTokens[address(0)][transferData.destChain] += msg.value;
267
268         TokenTransfer.Data memory packetData = TokenTransfer.Data({
269             srcChain: sourceChain,
270             destChain: transferData.destChain,
271             sender: msg.sender.addressToString(),
272             receiver: transferData.receiver,
273             amount: msg.value.toBytes(),
274             token: address(0).addressToString(),
275             oriToken: ""
276         });
277
278         // send packet
279         string[] memory ports = new string[](1);
```

```
280         bytes[] memory dataList = new bytes[](1);
281         ports[0] = PORT;
282         dataList[0] = TokenTransfer.encode(packetData);
283         PacketTypes.Packet memory crossPacket = PacketTypes.Packet({
284             sequence: packet.getNextSequenceSend(
285                 sourceChain,
286                 transferData.destChain
287             ),
288             sourceChain: sourceChain,
289             destChain: transferData.destChain,
290             relayChain: transferData.relayChain,
291             ports: ports,
292             dataList: dataList
293         });
294         packet.sendPacket(crossPacket);
295     }
```

Listing 2.1: Transfer.sol

Field Name	Description
srcChain	The chain to launch the cross-chain transfer
destChain	The chain to receive and execute the cross-chain transfer
sender	The account address of sender (in srcChain)
receiver	The account address of receiver (in destChain)
amount	The amount of transferred tokens
token	The token address in srcChain
oriToken	If token is a xibc-token, it's the token address in origin chain, otherwise, it's null

Table 2.2: The metadata of cross-chain transfer

`sendTransferBase` The user can invoke this interface to launch a cross-chain transfer that asks the user to lock a certain amount of native coins into the *Transfer* contract. After that it packs the transfer metadata into the cross-chain packet and emits it. There are two questions we need to answer:

1. Is it possible to let *Transfer* emit cross-chain events without locking money?
   => **No**, because the check in line 264 that requires the caller to transfer native coins.
2. Is it possible to mislead *Transfer* to pack wrong transfer metadata?
   => To answer this question, we should list all metadata first, and then find out where their values come from. Specifically, the fields of cross-transfer metadata are shown in Table 2.2. According to the assignment code in line 268 to line 276, the caller can manipulate two fields: `destChain` and `receiver`. The caller can corrupt the two fields to wrong values, which make the *Transfer* emit a cross-chain packet with a wrong `destChain` or `receiver`. If the `destChain` is wrong, then the cross-chain packet will be forwarded by the Teleport chain to all supported chains, which will forward it again due to the wrong `destChain`. Reasonably, the above process will only happen once, because all clients check if the cross-packet has been received. Finally, the packet will be thrown away by relayers. Furthermore, the wrong `receiver` will make the caller's money irretrievable. **Therefore, although the caller can let *Transfer* to pack wrong transfer metadata, there is no harm to the Teleport project.**

`sendTransferERC20` The difference between this function with above is that this function asks the user to lock ERC20 tokens. The ERC20 tokens maybe two types of tokens. The first one is the normal ERC20

token, such as USDC and WETH, while another one is the xibc-token that is the cross-chain token. If the token is a cross-chain token, *Transfer* (in `srcChain`) will burn the caller's xibc-token and *Transfer* (in `destChain`) will unlock `receiver`'s original tokens. When confirming the above two questions, we focus on the field of `oriToken` especially. After that, we find that a wrong cross-chain packet will be executed successfully in `destChain` if there are tokens with the same address on different chains. That may cause money losses to the project. Since the odds are low, we put it in the recommendation Section 3.3.1.

Field Name	Description
`srcChain`	The chain to launch the cross-chain invocation
`destChain`	The chain to receive and execute the cross-chain invocation
`sender`	The account address of sender (in `srcChain`)
`contractAddress`	The callee contract address (in `destChain`)
`data`	The payload of the cross-chain contract invocation

Table 2.3: The metadata of cross-chain contract invocation

`sendRemoteContractCall` The user can invoke this interface to launch a cross-chain contract invocation. As shown in the codes, it barely checks inputs and charges no fees. We present a potential DoS attack method for that in Section 3.2.1. Besides, we also list the metadata of a cross-chain invocation in Table 2.3, and we notice that `destChain`, `contractAddress`, and `data` are indicated by the caller.

```
119    function onRecvPacket(bytes calldata data)
120        external
121        override
122        onlyPacket
123        returns (PacketTypes.Result memory)
124    {
125        RemoteContractCall.Data memory packetData = RemoteContractCall.decode(
126            data
127        );
128
129        require(
130            packetData.contractAddress.parseAddr() != address(this),
131            "illegal operation"
132        );
133
134        require(
135            packetData.contractAddress.parseAddr() != address(packet),
136            "illegal operation"
137        );
```

Listing 2.2: RCC.sol

Similarly, there are three questions we need to answer:

1. Is it possible to transfer out the Teleport project's assets by cross-chain invoking the `transfer` function? => We first check two cases for this question. The first one is that the `transfer` function is a legal cross-chain invocation, and the second one is that the executor in `destChain` is the *RCC* contract deployed on `destChain`. Theoretically, all ERC-20 token balances in *RCC* can be withdrawn by anyone via the cross-chain invocation. We then go through all codes of this contract and find that *RCC* do not store any assets by design. Certainly, it's okay to donate ERC-20 tokens to *RCC*, while the money can be taken away by anyone. Therefore, the answer is **no**.

2. Is it possible to perform privileged operations in the Teleport project via a cross-chain invocation?
   => To answer this question, we first locate which contracts (in the Teleport project) that *RCC* invokes, We find that the only contract it invokes is the *Packet*. However, the cross-chain invocation to *Packet* is illegal, as shown in the `onRecvPacket` function (line 134 to line 137). The answer is also **no**.
3. Is it possible to send an endless cross-chain transaction that invokes the `sendRemoteContractCall` function of *RCC* (in `destChain`)?
   => **No**, because the cross-chain invocation to *RCC* is also illegal, as shown in the `onRecvPacket` function (line 129 to line 132).

```
65    uint256 remainingValue = msg.value;
66    for (uint64 i = 0; i < multiCallData.functions.length; i++) {
67        require(multiCallData.functions[i] < 3, "invlaid function ID");
68        if (multiCallData.functions[i] == 0) {
69            MultiCallDataTypes.ERC20TransferData memory data = abi.decode(
70                multiCallData.data[i],
71                (MultiCallDataTypes.ERC20TransferData)
72            );
73            dataList[i] = callTransferERC20(multiCallData.destChain, data);
74            ports[i] = "FT";
75        } else if (multiCallData.functions[i] == 1) {
76            MultiCallDataTypes.BaseTransferData memory data = abi.decode(
77                multiCallData.data[i],
78                (MultiCallDataTypes.BaseTransferData)
79            );
80            require(data.amount > 0, "invalid amount");
81            require(remainingValue >= data.amount, "invalid value");
82            dataList[i] = callTransferBase(multiCallData.destChain, data);
83            ports[i] = "FT";
84            remainingValue -= data.amount;
85        }
```

Listing 2.3: MultiCall.sol

`multiCall` The user can invoke this function to pack multiple cross-chain actions (including transfers and invocations) into one cross-chain transaction. According to the features of multiple calls, we have two concerns need to check:

1. How does *MultiCall* handle `msg.value` when the multi-call contains multiple transfers of native coins?
   => As shown in the codes (line 65 and line 84), *MultiCall* does not use `msg.value` in a loop. Indeed it uses a variable `remainingValue` to record the remaining native coins, which is a **safe** way.
2. Does *MultiCall* store money? Does *MultiCall* execute cross-chain invocations? If so, the caller can transfer the reserves of *MultiCall*.
   => The answer is **no** and **no**. The core task of *MultiCall* is packing multicall packet, and all assets are forwarded into *Transfer*. The multicall packet will be split (in `destChain`) for *Transfer* and *RCC* to execute. This design has **no risk**.

2.3 Interfaces for accounts in Teleport chain

Since the business logic of the four interfaces: `sendTransferBase`, `sendTransferERC20`, `sendRemoteContractCall`, and `multiCall` in Teleport chain is same with them in EVM compatible chain, we follow the

same thought to audit the four interfaces. After that, we do not find any issues in them, except for the lack of fee mechanism (in Section 3.2.1).

2.4 Interfaces for relayers

`recvPacket` (Solidity) A relayer (or someone else) invokes this interface to forward a cross-chain packet from the Teleport chain to the *Packet* contract in each supported chain. It then verifies and executes the cross-chain packet. After that, it emits an acknowledgment event to tell the source chain the result of the cross-chain transaction.

Field Name	Description
`sourceChain`	The chain to emit to cross-chain packet
`destChain`	The chain to receive and execute the cross-chain packet
`relayChain`	The relay chain to forward the cross-chain packet
`ports`*	"FT" or "Contract" represents the cross-chain transfer or invocation
`dataList`*	A list of cross-chain actions
`sequence`	The sequence of the cross-chain packet, which recorded in `sourceChain` and `relayChain`

Table 2.4: Fields of a cross-chain packet

*: The length of the two lists (`ports` and `dataList`) is the same, and each item in `ports` represents the cross-chain action type of the corresponding item in `dataList`.

```
345    IClient client;
346    if (
347        Strings.equals(destChain, clientManager.getChainName()) &&
348        bytes(relayChain).length > 0
349    ) {
350        client = clientManager.getClient(relayChain);
351    } else {
352        client = clientManager.getClient(sourceChain);
353    }
354    require(address(client) != address(0), "light client not found!");
355
356    client.verifyPacketCommitment(
357        sender,
358        height,
359        proof,
360        sourceChain,
361        destChain,
362        sequence,
363        commitBytes
364    );
```

Listing 2.4: Packet.sol

We look into this function to answer the following two questions:

1. Which parameters of `recvPacket` can be corrupted, and what's the impact of the corruption?
   => The function `recvPacket` has three parameters: `packet`, `proof` and `height`. We list all fields of a packet in Table 2.4. Among them, `proof` and `height` are used to verify the existence of `packet.dataList` in the Teleport chain. Furthermore, `packet.sourceChain`, `packet.destChain`, and `sequence` construct a

path "commitments/sourceChain/destChain/sequence" to trace the `packet.dataList` from the Merkle tree of the Teleport chain. Therefore, modifying these values can not pass the verification process. And then we look at the remaining two values: `packet.relayChain` and `packet.ports`. Modifying the first one can not pass the pre-check, while changing another one can cause a potential DoS attack, which are shown in Section 3.2.2.

2. How does *Packet* verify the existence of `packet.dataList`?
   => Most of blockchain systems (including the Teleport chain) uses the Merkle tree structure for fast integrity verification. That's because Merkle tree is a traceable structure that allows efficient and secure verification of contents of a large data structure. *Packet* has the ability to verify the contents of the Merkle tree as long as the root is stored in advance. Particularly, the verification to `packet.dataList` depends on two: the first is that registered relayers forward the state root of the source chain in advance, and the second is that callers of `recvPacket` pass the correct parameters: `proof` and `height`. The former is all intermediate nodes necessary for verify `packet.dataList`, and the latter is used to specify the state root of which block header.

`acknowledgement` (Solidity) A relayer (or someone else) invokes this function to forward the execution result of a cross-chain packet from the destination chain back to the source chain. The whole process is quite similar with the function `recvPacket`, but it verifies the existence of the acknowledgement of the cross-chain packet. We follow the same thought to audit this function, and find it also has the issue described in Section 3.2.2.

`updateClient` (Solidity) The function `updateClient` is a privileged function that can be accessed by only registered relayers. These registered relayers always monitor and forward newly appended block headers from the Teleport chain to supported chains and from all supported chains to the Teleport chain. Particularly, they invoke `updateClient` of the *Tendermint* contract in supported chains to forward headers from the Teleport chain. The question is how *Tendermint* trusts (or verifies) a forwarded block header if the registered relayer does not do bad thing. After reading the codes, we summarize the whole verification thought as that:

1. The forwarded header contains signatures signed by a group of validators, which covers the whole tendermint block header.
2. Each tendermint header stores the hash of validators that can produce the next block, and *Tendermint* stores the validators's hash.
3. Therefore, if the hash of validators signing the forwarded block header equals to the stored validators' hash, *Tendermint* will trust the forwarded header.

The validation method is make sense for the PoS chain, but the concern is that is it possible to modify the value of stored validators' hash by other means, such an authorized function that may be compromised. That's because the value is critical for the verification. We find that there are only three functions to change the value, which are `updateClient`, `createClient`, and `upgradeClient`. Particularly, the last two can be accessed by only `CREATE_CLIENT_ROLE` and `UPGRADE_CLIENT_ROLE`. Therefore, the reliability of the header verification depends on the protection to the private keys of these privileged EOA. Besides, during the auditing, we find that a critical code missing that will cause the header verification useless, which shown in Section 3.1.1.

`RecvPacket`, `Acknowledgement`, **and** `UpdateClient` (Golang) The *xibc* module of the Teleport chain implements three light clients: eth, bsc, and tendermint, which all have the three functions. The purposes of

them are same with that of `recvPacket`, `acknowledgement`, and `updateClient` (in Solidity), but the implementation are different due to the different header structure and the consensus mechanism.

Since the similar solution (Merkle tree), they all verify cross-chain packet and ack-packet by tracing the state root. For the verification to forwarded headers, they act like a real light node in Ethereum, BSC, and a tendermint chain. Therefore, the audit thought is to compare them with the official light nodes.

Besides, these Golang codes do not verify `packet.ports`, and is susceptible to the issue in Section 3.2.2.

2.5 Interfaces for the Teleport project

There are some privileged interfaces that are critical to the Teleport project, as shown in Table 2.1. The access to these interfaces in *xibc* module (in Golang) is controlled by the proposal mechanism of *gov* module [2], which is decentralized design. However, the access to interfaces in Solidity is limit to some authorized EOAs. We have a recommendation for that in Seciton 3.3.3.

Since these functions are privileged, our audit mainly focus on discovering their implementation or design mistakes.

[2] https://github.com/cosmos/cosmos-sdk/tree/master/x/gov

Chapter 3 Findings

In total, we find three potential issues in the project. We also have three recommendations, as follows:

- High Risk: 2
- Medium Risk: 1
- Low Risk: 0
- Recommendations: 3

ID	Severity	Description	Category	Status
1	High	*The lack of* `Ed25519.verify` *codes*	Software Security	Fixed
2	Medium	*A potential DoS attack I*	DeFi Security	Confirmed
3	High	*A potential DoS attack II*	DeFi Security	Fixed
4	-	*Add a check to ensure the* `oriChain` *equals to* `destChain`	Recommendation	Fixed
5	-	*Fix typos*	Recommendation	Fixed
6	-	*Address the concern of the centralization design*	Recommendation	Confirmed

The details are provided in the following sections.

3.1 Software Security

3.1.1 The lack of `Ed25519.verify` codes

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the verification process to the headers forwarded by registered relayers, the most critical part is to check if the signatures in headers are signed by current validators (stored in the *Tendermint* contract), which is the task of `Ed25519.verify`. However, the implementation of this function is null.

```
303    // Validate signature.
304    bytes memory signBytes = genVoteSignBytes(commit, chainID, i);
305    Ed25519.verify(
306        val.pub_key.ed25519,
307        signBytes,
308        commit.signatures[i].signature
309    );
```

Listing 3.1: LightClient.sol

```
 6    library Ed25519 {
 7        //TODO
 8        function verify(
 9            bytes memory pubkey,
10            bytes memory sig,
11            bytes memory signBytes
12        ) internal pure {}
13    }
```

Listing 3.2: Ed25519.sol

Impact The verification to forwarded headers can be easily bypassed.

Suggestion Supply the codes of `Ed25519.verify`.

3.2 DeFi Security

3.2.1 A potential DoS attack I

Status Confirmed

Introduced by `Version 1`

Description Since the Teleport project does not charge fees to launch a cross-chain transaction, the cost to launch a cross-chain contract invocation is only the transaction fees charged by miners (or validators) in blockchain. The transaction fees are different on different chains. For example, transaction fees in Ethereum are higher than that on the BSC.

If an attacker continuously launches cross-chain invocations from BSC to Ethereum, relayers between Teleport chain and Ethereum will forward a lot of cross-chain packets to Ethereum, which are charged by more expensive fees than attack cost. Although the attacker can not profit from that, relayers between Teleport chain and Ethereum will suffer from the loss of the transaction fee.

```
57    function sendRemoteContractCall(RCCDataTypes.RCCData calldata rccData)
58        external
59        override
60    {
61        string memory sourceChain = clientManager.getChainName();
62        require(
63            !sourceChain.equals(rccData.destChain),
64            "sourceChain can't equal to destChain"
65        );
66
67        RemoteContractCall.Data memory packetData = RemoteContractCall.Data({
68            srcChain: sourceChain,
69            destChain: rccData.destChain,
70            sender: msg.sender.addressToString(),
71            contractAddress: rccData.contractAddress,
72            data: rccData.data
73        });
74
75        // send packet
76        string[] memory ports = new string[](1);
77        bytes[] memory dataList = new bytes[](1);
78        ports[0] = PORT;
79        dataList[0] = RemoteContractCall.encode(packetData);
80        PacketTypes.Packet memory crossPacket = PacketTypes.Packet({
81            sequence: packet.getNextSequenceSend(
82                sourceChain,
83                rccData.destChain
84            ),
```

```
85              sourceChain: sourceChain,
86              destChain: rccData.destChain,
87              relayChain: rccData.relayChain,
88              ports: ports,
89              dataList: dataList
90          });
91          packet.sendPacket(crossPacket);
92      }
```

Listing 3.3: RCC.sol

Impact Relayers may stop working due to the high transaction fees caused by a potential DoS attack.

Suggestion Charge the launch of a cross-chain transaction.

Feedback from the Project Under the current design, the relayers do not forward cross-chain packages without paying enough fees. In order to realize the vision that anyone can forward cross-chain packages, we're designing the fee mechanism in the on-chain smart contracts.

3.2.2 A potential DoS attack II

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `recvPacket` does not verify if `packet.ports` is modified, and it executes the cross-chain packet (in line 264) after receiving it (in line 258).

```
226     function recvPacket(
227         PacketTypes.Packet calldata packet,
228         bytes calldata proof,
229         Height.Data calldata height
230     ) external override {
231         bytes memory packetReceiptKey = Host.packetReceiptKey(
232             packet.sourceChain,
233             packet.destChain,
234             packet.sequence
235         );
236
237         require(!receipts[packetReceiptKey], "packet has been received");
238
239         bytes memory dataSum;
240         for (uint64 i = 0; i < packet.ports.length; i++) {
241             dataSum = Bytes.concat(
242                 dataSum,
243                 Bytes.fromBytes32(sha256(packet.dataList[i]))
244             );
245         }
246
247         verifyPacketCommitment(
248             _msgSender(),
249             packet.sequence,
250             packet.sourceChain,
251             packet.destChain,
252             packet.relayChain,
```

```
253             proof,
254             height,
255             Bytes.fromBytes32(sha256(dataSum))
256         );
257
258         receipts[packetReceiptKey] = true;
259
260         emit PacketReceived(packet);
261
262         if (Strings.equals(packet.destChain, clientManager.getChainName())) {
263             Acknowledgement.Data memory ack;
264             try this.executePacket(packet) returns (bytes[] memory results) {
265                 ack.results = results;
266             } catch Error(string memory message) {
267                 ack.message = message;
268             }
269             bytes memory ackBytes = Acknowledgement.encode(ack);
270             writeAcknowledgement(
271                 packet.sequence,
272                 packet.sourceChain,
273                 packet.destChain,
274                 packet.relayChain,
275                 ackBytes
276             );
277
278             emit AckWritten(packet, ackBytes);
279         } else {
280             require(
281                 address(clientManager.getClient(packet.destChain)) !=
282                     address(0),
283                 "light client not found!"
284             );
285             commitments[
286                 Host.packetCommitmentKey(
287                     packet.sourceChain,
288                     packet.destChain,
289                     packet.sequence
290                 )
291             ] = sha256(dataSum);
292
293             emit PacketSent(packet);
294         }
295     }
```

Listing 3.4: Packet.sol

Furthermore, the purpose of `packet.ports` is choosing a contract (in line 308) to execute the corresponding cross-chain action. For example, the *Transfer* contract can execute cross-chain transfer with the port as "FT", while the *RCC* contract can execute cross-chain invocation with the port as "CONTRACT".

```
301     function executePacket(PacketTypes.Packet calldata packet)
302         external
303         onlySelf
304         returns (bytes[] memory)
```

```
305     {
306         bytes[] memory results = new bytes[](packet.ports.length);
307         for (uint64 i = 0; i < packet.ports.length; i++) {
308             IModule module = routing.getModule(packet.ports[i]);
309             require(
310                 address(module) != address(0),
311                 Strings.uint642str(i).toSlice().concat(
312                     ": module not found!".toSlice()
313                 )
314             );
315             PacketTypes.Result memory res = module.onRecvPacket(
316                 packet.dataList[i]
317             );
318             require(
319                 res.result.length > 0,
320                 Strings
321                     .uint642str(i)
322                     .toSlice()
323                     .concat(": ".toSlice())
324                     .toSlice()
325                     .concat(res.message.toSlice())
326             );
327             results[i] = res.result;
328         }
329         return results;
330     }
```

Listing 3.5: Packet.sol

Therefore, the cross-chain packet with wrong `packet.ports` can pass the packet verification but can not be executed successfully. Furthermore, since the packet is received (in line 264), the packet with correct `packet.ports` will be rejected by the *Packet* contract (in line 237).

As a result, if an attacker monitors all cross-chain packets and continuously modifies their `packet.ports` as wrong values, the Teleport project will be paralyzed.

Impact The Teleport project may be paralyzed by a potential DoS attack.

Suggestion Add `packet.ports` to the cross-chain packet verification (for `verifyPacketCommitment` and `verifyPacketAcknowledgement` in Solidity and Golang).

3.3 Additional Recommendation

3.3.1 Add a check to ensure the `oriChain` equals to `destChain`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The *Transfer* contract uses a mapping `bindings` to record the mapping relationship from xibc-tokens to original tokens. Each token corresponds to four fields:

- `oriChain`: is the original chain.
- `oriToken`: is the original address in `oriChain`.

- `amount`: is the amount of `oriToken`s transferred from `oriChain` to this chain.
- `bound`: is a boolean that identifies this mapping relationship is valid.

We notice that *Transfer* does not check if `oriChain` equals to `destChain` when a user invokes the `sendTransferERC20` to send back tokens to `oriChain`. That's maybe a potential low risk that will be illustrated by the following attack method.

There is a very unlikely but theoretically possible assumption: token A in Chain A and token B in Chain B share the same contract address, and the Teleport project supports these two tokens.

The attacker launches a cross-chain transfer to send back A tokens to Chain A but set the `destChain` as Chain B. Since the lack of verification, *Transfer* emits the cross-chain packet.

```
122    packetData = TokenTransfer.Data({
123        srcChain: sourceChain,
124        destChain: transferData.destChain,
125        sender: msg.sender.addressToString(),
126        receiver: transferData.receiver,
127        amount: transferData.amount.toBytes(),
128        token: transferData.tokenAddress.addressToString(),
129        oriToken: bindings[transferData.tokenAddress].oriToken
130    });
```

Listing 3.6: Transfer.sol

Then, we see the `onRecvPacket` function that checks if the `amount` less than its recorded amount of `oriToken` going to `srcChain` (in line 372 to line 375) before transferring `oriToken` (in line 384). Now, the cross-chain packet is forwarded to chain B. The check may be passed due to the assumption, then *Transfer* transfers `amount` token B to the attacker's account(`receiver`) in Chain B. If the price of token B is higher than token A, the attacker will profit from the attack (he swaps xibc-tokenA for tokenB).

```
370    } else if (packetData.oriToken.parseAddr() != address(0)) {
371        // ERC20 token back to origin
372        if (
373            packetData.amount.toUint256() >
374            outTokens[packetData.oriToken.parseAddr()][packetData.srcChain]
375        ) {
376            return
377                _newAcknowledgement(
378                    false,
379                    "onRecvPackt: amount could not be greater than locked amount"
380                );
381        }
382
383        if (
384            !IERC20(packetData.oriToken.parseAddr()).transfer(
385                packetData.receiver.parseAddr(),
386                packetData.amount.toUint256()
387            )
388        ) {
389            return
390                _newAcknowledgement(
391                    false,
392                    "onRecvPackt: unlock to receiver failed"
393                );
```

```
394         }
395
396         outTokens[packetData.oriToken.parseAddr()][
397             packetData.srcChain
398         ] -= packetData.amount.toUint256();
399     }
```

Listing 3.7: Transfer.sol

Impact An attack method that exists in a theoretical situation will cause losses to the project.

Suggestion Add a check to ensure the `oriChain` equals to `destChain` in the `sendTransferERC20` and `transferERC20` functions of EVM contracts.

3.3.2 Fix typos

Status Fixed in `Version 2`.

Introduced by `Version 1`

Description

```
67  require(multiCallData.functions[i] < 3, "invlaid function ID");
```

Listing 3.8: MultiCall.sol

Impact NA.

Suggestion Fix the typo.

3.3.3 Address the concern of the centralization design

Status Confirmed.

Introduced by `Version 1`

Description There are a lot of privileged functions in Solidity that are critical to the project. For example, the `upgrade` function can reset the current state root and validators's hash of the *Tendermint* contract, which can directly affect the verification of the following headers and cross-chain packets. If the private keys of these EOA are leaked, the whole project will be compromised, and the user's money locked in *Transfer* contract will at risk.

Impact The project has the authority to withdraw all reserves locked in the cross-chain bridge (*Transfer*), and has the risk of being attacked due to the private key leak.

Suggestion Adopt a decentralized method to invoke these privileged functions in Solidity, and leverage a secure private key solution (e.g., multi-signed wallet, and TEE based security key management) to manage the private key of privileged EOAs.

Feedback from the Project We will transfer the admin role to the Gnosis safe multi-signature wallet to keep the safety of privileged account.