

HAECHI AUDIT

Iskra Bridge

Smart Contract Security Analysis

Published on : Nov 11, 2022

Version v1.1





HAECHI AUDIT

Smart Contract Audit Certificate




IskraBridge

Security Report Published by HAECHI AUDIT

v1.1 Nov 11, 2022 - add final commit

v1.0 september 30, 2022 - final report

Auditor : Jinu Lee, Jade Han

 *hejung han*

Executive Summary

Severity of Issues	Findings	Resolved	Unresolved	Acknowledged	Comment
Critical	-	-	-	-	-
High	-	-	-	-	-
Medium	1	1	-	-	-
Low	-	-	-	-	-
Tips	2	1	-	1	-

TABLE OF CONTENTS

[TABLE OF CONTENTS](#)

[ABOUT US](#)

[INTRODUCTION](#)

[SUMMARY](#)

[Summary of Audit Scope](#)

[Summary of Findings](#)

[OVERVIEW](#)

[Audit Scope](#)

[Access Control](#)

[Project Overview](#)

[Notice](#)

[FINDINGS](#)

[1. When moving assets between chains, user can avoid paying serviceFee by calling returnTransfer twice.](#)

[2. Improper positioning of user input validation.](#)

[3. gas optimization tips](#)

[Fix](#)

[Fix Comment](#)

[DISCLAIMER](#)

ABOUT US

HAECHI AUDIT believes in the power of cryptocurrency and the next paradigm it will bring.

We have the vision to empower the next generation of finance. By providing security and trust in the blockchain industry, we dream of a world where everyone has easy access to blockchain technology.

HAECHI AUDIT is a flagship service of HAECHI LABS, the leader of the global blockchain industry. HAECHI AUDIT provides specialized and professional smart contract security auditing and development services.

We are a team of experts with years of experience in the blockchain field and have been trusted by 400+ project groups. Our notable partners include Sushiswap, 1inch, Klaytn, Badger, etc.

HAECHI AUDIT is the only blockchain technology company selected for the Samsung Electronics Startup Incubation Program in recognition of our expertise. We have also received technology grants from the Ethereum Foundation and Ethereum Community Fund.

Inquiries: audit@haechi.io

Website: audit.haechi.io

INTRODUCTION

This report was prepared to audit the security of the Iskra Bridge smart contract created by the Iskra team. HAECHI AUDIT conducted the audit focusing on whether the smart contract created by Iskra team is soundly implemented and designed as specified in the published materials, in addition to the safety and security of the smart contract.

In detail, we have focused on the following as requested by Iskra Bridge

- Upgradeable Contract Issues
 - Signature Replay
 - Bridge Logic
 - Token Bridging Model
-

HAECHI AUDIT recommends the Iskra team improve all issues discovered.

SUMMARY

Summary of Audit Scope

The codes used in this Audit can be found at GitHub

(<https://github.com/iskraworld/iskra-contracts/releases/tag/v1.2.0-bridgealpha.3>)

The last commit used for Audit is 5f77c1b9dfbb746f4e79fe09ce49d29f5f96c9b7.

Summary of Findings

HAECHI LABES found 1 issue (1 Medium, 2 Tips)

#ID	Title	Type	Severity	Difficulty
1	When moving assets between chains, user can avoid paying serviceFee by calling returnTransfer twice	Logic Bug	Medium	Low
2	Improper positioning of user input validation	Input Validation	Tips	N/A
3	Gas Optimization TIPS	Gas Optimization	Tips	N/A

OVERVIEW

Audit Scope

- └─ **bridge**
 - └─ **adapter**
 - └─ TokenAdapterImplementation.sol
 - └─ TokenAdapterState.sol
 - └─ **beacon**
 - └─ Beacon.sol
 - └─ BeaconProxyCreator.sol
 - └─ **governance**
 - └─ BridgeGovernanceMessage.sol
 - └─ IskraBridgeGovernance.sol
 - └─ **interface**
 - └─ IBeaconProxyCreator.sol
 - └─ IBurnable.sol
 - └─ IFeePolicy.sol
 - └─ IGovernance.sol
 - └─ IMessenger.sol
 - └─ IMintable.sol
 - └─ **utils**
 - └─ ERC20Querier.sol
 - └─ **wormhole**
 - └─ Getters.sol
 - └─ Governance.sol
 - └─ GovernanceStructs.sol
 - └─ Implementation.sol
 - └─ Messages.sol
 - └─ Migrations.sol
 - └─ Setters.sol
 - └─ Setup.sol
 - └─ State.sol
 - └─ Structs.sol
 - └─ Wormhole.sol
 - └─ **bridge**
 - └─ Bridge.sol

- | | | | BridgeGetters.sol
- | | | | BridgeGovernance.sol
- | | | | BridgeImplementation.sol
- | | | | BridgeSetters.sol
- | | | | BridgeSetup.sol
- | | | | BridgeState.sol
- | | | | BridgeStructs.sol
- | | | | FixedRatioFee.sol
- | | | | TokenBridge.sol
- | | | | **token**
- | | | | | Token.sol
- | | | | | TokenImplementation.sol
- | | | | | TokenState.sol
- | | | | **utils**
- | | | | **interfaces**
- | | | | | IWormhole.sol
- | | | | **libraries**
- | | | | | **external**
- | | | | | BytesLib.sol
- | | | **multisig**
- | | | | MultiSigContract.sol
- | | | | MultiSigContractGovernable.sol
- | | | | MultiSigWallet.sol

We note that Relayer and Guardian code are not part of audit scope.

Access Control

Iskra Bridge contracts have the following access control mechanisms.

- ❖ `onlyOperator()`
- ❖ `onlyOwner()`
- ❖ `onlyVoter()`
- ❖ `onlyProposer()`
- ❖ `verifyGovernanceVM()`
- ❖ `verifyVM()`
- ❖ `verifyBridgeVM()`
- ❖ `onlySelfCall()`

onlyOperator() : modifier that controls access related to `TokenAdapterImplementation` mint/burn functions.

- ❖ `TokenAdapterImplementation#mint`
- ❖ `TokenAdapterImplementation#burn`

onlyOwner() : modifier that controls access related to `TokenImplementation` mint/burn functions.

- ❖ `TokenImplementation#mint`
- ❖ `TokenImplementation#burn`

onlyVoter() : modifier that controls access related to proposal voting action in `MultiSigContract`.

- ❖ `MultiSigContract#confirmTransaction`
- ❖ `MultiSigContract#revokeTransaction`
- ❖ `MultiSigContract#executeTransaction`

onlyProposer() : modifier that controls access related to updating proposal in `MultiSigContract` and `MultiSigContractGovernable`

- ❖ `MultiSigContract#submitTransaction`
- ❖ `MultiSigContractGovernable#proposeAddVoter`
- ❖ `MultiSigContractGovernable#proposeRemoveVoter`
- ❖ `MultiSigContractGovernable#proposeAddProposer`
- ❖ `MultiSigContractGovernable#proposeRemoveProposer`
- ❖ `MultiSigContractGovernable#proposeChangeQuorumSize`
- ❖ `MultiSigContractGovernable#proposeCheckpoint`

verifyGovernanceVM() : function to verify that payload issued by Governance chain.

- ❖ `Governance#submitContractUpgrade`

- ❖ Governance#submitSetMessageFee
- ❖ Governance#submitNewGuardianSet
- ❖ Governance#submitTransferFees
- ❖ Governance#submitRecoverChainId
- ❖ Bridge#createWrapped
- ❖ Bridge#createAdapter
- ❖ BridgeGovernance#registerChain
- ❖ BridgeGovernance#upgrade
- ❖ BridgeGovernance#updateServiceFeePolicy

verifyVM() : function to verify that payload is signed by Guardian sets.

- ❖ IskraBridgeGovernance#createWrapped
- ❖ IskraBridgeGovernance#createAdapter
- ❖ Governance#verifyGovernanceVM
- ❖ Messages#parseAndVerifyVM
- ❖ Bridge#_parseAndVerifyVM
- ❖ BridgeGovernance#verifyGovernanceVM

verifyBridgeVM() : function to verify that payload has correct chain id and emitterAddress.

- ❖ Bridge#_verifyTransferVM

onlySelfCall() : modifier to verify that msg.sender is self

- ❖ IskraBridgeGovernance#executeChangeConsistencyLevel
- ❖ IskraBridgeGovernance#executeSetVerifier
- ❖ MultiSigContractGovernable#addVoter
- ❖ MultiSigContractGovernable#removeVoter
- ❖ MultiSigContractGovernable#addProposer
- ❖ MultiSigContractGovernable#removeProposer
- ❖ MultiSigContractGovernable#changeQuorumSize
- ❖ MultiSigContractGovernable#checkpointTransactionId

Project Overview

This project is derived from Wormhole Bridge, and We carefully observed code about other parts between Wormhole Bridge Service and Iskra Bridge Service. Asset transferring between chains, one of the main functions of the project, is handled similarly to the Wormhole Bridge Service.

Iskra Bridge is a bridge that exchanges various assets between Ethereum, Klaytn, and Amethyst chains. Only ERC20 assets approved by Iskra Governance can be transferred through Iskra Bridge. ERC20 owners can create Wrapped Token to transfer between other chains through Iskra Bridge Service. Original Wormhole Bridge Service supports asset custody in Bridge Contract everyone can create wrapped tokens and transfer tokens between chains through Bridge Contract. On the other side, Iskra Bridge Service manages Wrapped Token using TokenAdapter, not Bridge Contract. Wrapped Token is created using Only tokens approved by Governance. In Iskra Bridge Service, the structure of Beacon Proxy is changed differently from the Original Wormhole Bridge Service because of TokenAdapter management. When a user publishes a message, the message publishing layer is different depending on the kind of source chain. When the user transfers an asset from Ethereum or Klaytn to Amethyst, the Original Wormhole Core Contract is used as a Message Publish Layer. Conversely, When transferring assets from Amethyst to Ethereum or Klaytn, the Wormhole Core Contract modified by Iskra is used as a Message Publishing Layer. Unlike Original Wormhole Bridge, users who want to transfer assets should pay an additional Service Fee. IskraBridgeGovernance handles the Iskra bridge service operation tasks, including policy updating and contract updating.

Notice

This audit scope included only contracts used in Iskra Bridge. Off-chain elements such as Guardian code and Relayer code used in Iskra Bridge are not included in this audit scope.

FINDINGS

1. When moving assets between chains, user can avoid paying serviceFee by calling returnTransfer twice.

ID: ISKRA-BRIDGE-01

Severity: Medium

Type: Logic Bug

Difficulty: Low

File: bridge/wormhole/bridge/Bridge.sol

Impact

In Iskra Bridge, users can use Bridge Service without paying service fee.

Iskra may suffer continuous economic damage because of failing to collect a service fee.

Description

Unlike Wormhole Bridge, Iskra Bridge add serviceFee policy and returnTransfer function.

- Service Fee Policy

Unlike Wormhole Bridge, Iskra Bridge collect service fee in addition to arbiter fee.

service fee is occurred by iskra bridge token transfer process, is saved in Iskra FeeCollector.

- *returnTransfer* function

In Wormhole Bridge, even if token that has not been created into wrapped token is sent, the transferred token is not frozen because anyone can create Wrapped Token.

However, Iskra Bridge Users cannot create Wrapped Token without Governance approval.

so, sending Token that has not been created with Wrapped Token may freeze until

Wrapped Token is created by Governance approval.

To prevent this scenario, Iskra team added *returnTransfer* function that cancels token transfer process in Iskra Bridge.

returnTransfer function that cancels token transfer process updates payload service fee to zero.

```

function returnTransfer(
    bytes memory encodedVm,
    bytes32 recipient,
    uint256 arbiterFee,
    uint32 nonce
) public payable nonReentrant returns (uint64 sequence) {
    ...
    sequence = logTransfer(
        transfer.tokenChain,
        transfer.tokenAddress,
        transfer.amount,
        vm.emitterChainId,
        recipient,
        normalizeAmount(arbiterFee, decimals),
        0, // ← set serviceFee 0
        msg.value,
        nonce
    );
}

```

<https://github.com/iskraworld/iskra-contracts/blob/v1.2.0-bridgealpha.3/contracts/bridge/wormhole/bridge/Bridge.sol#L517-L582>

For convenience, we will refer to source chain as “A Chain” and destination chain as “B Chain”.

If user execute normal token transfer cancel flow, process is as below.

transferTokens(A Chain) -> returnTransfer(B Chain) -> completeTransfer(A Chain)

However, if malicious user calls returnTransfer function on Both “A Chain” and “B Chain” using below process, malicious user can bypass paying service fee.

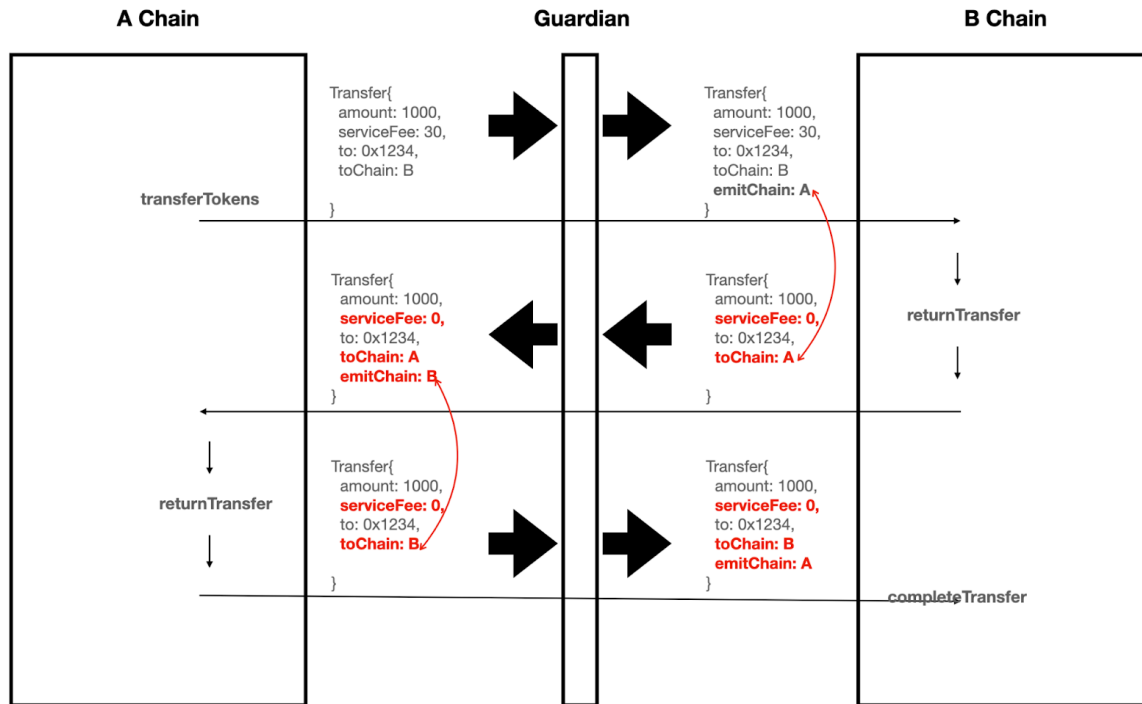
transferTokens(A Chain) -> returnTransfer(B Chain) -> returnTransfer(A Chain) -> completeTransfer(B Chain)

Payload generated from returnTransfer function calling in specific chain can be used again in returnTransfer function calling of another chain.

Because payload created by transferToken function calling and Payload created by *returnTransfer* function calling are the same format.

For above reasons, Malicious users can use Iskra Bridge Service without paying service fees.

mentioned process is illustrated in figure below.



Exploit Scenario

The reason this Scenario is possible is that the *completeTransfer* function handles VAA from *returnTransfer* function and VAA from *transferTokens* function the same.

Recommendation

- When emitting event in *returnTransfer* function, Add and Use *LogReturnTransfer* function that sets payloadID as 4. In the *returnTransfer* function, Add code to revert when the parsed payloadID is 4.

```
function logReturnTransfer(
    uint16 tokenChain,
    bytes32 tokenAddress,
    uint256 amount,
    uint16 recipientChain,
    bytes32 recipient,
    uint256 arbiterFee,
    uint256 serviceFee,
    uint256 callValue,
    uint32 nonce
) internal returns (uint64 sequence) {
    require(arbiterFee + serviceFee <= amount, "fee exceeds amount");
```

```

    BridgeStructs.Transfer memory transfer = BridgeStructs.Transfer({
        payloadID: 4,
        amount: amount,
        tokenAddress: tokenAddress,
        tokenChain: tokenChain,
        to: recipient,
        toChain: recipientChain,
        arbiterFee: arbiterFee,
        serviceFee: serviceFee
    });
    bytes memory encoded = encodeTransfer(transfer);
    sequence = wormhole().publishMessage{value: callValue}(nonce, encoded,
finality());
}

function returnTransfer(
    bytes memory encodedVm,
    bytes32 recipient,
    uint256 arbiterFee,
    uint32 nonce
) public payable nonReentrant returns (uint64 sequence) {
    (IWormhole.VM memory vm, BridgeStructs.Transfer memory transfer) =
_verifyTransferVM(encodedVm);

    require(transfer.payloadID!=4, "duplicate returnTransfer VAA");
    ...
    sequence = logReturnTransfer(
        transfer.tokenChain,
        transfer.tokenAddress,
        transfer.amount,
        vm.emitterChainId,
        recipient,
        normalizeAmount(arbiterFee, decimals),
        0,
        msg.value,
        nonce
    );
}

function returnTransfer(
    bytes memory encodedVm,
    bytes32 recipient,
    uint256 arbiterFee,
    uint8 feeDecimals,
    uint32 nonce
) public payable nonReentrant returns (uint64 sequence) {
    (IWormhole.VM memory vm, BridgeStructs.Transfer memory transfer) =
_verifyTransferVM(encodedVm);

```

```

require(transfer.payloadID!=4, "duplicate returnTransfer VAA");
...
sequence = logReturnTransfer(
    transfer.tokenChain,
    transfer.tokenAddress,
    transfer.amount,
    vm.emitterChainId,
    recipient,
    normalizeAmount(arbiterFee, feeDecimals),
    0,
    msg.value,
    nonce
);
}

```

- In `_parseTransferCommon` function, update the if statement to process when payloadID is 4.

```

function _parseTransferCommon(bytes memory encoded) public pure returns
(BridgeStructs.Transfer memory transfer) {
    uint8 payloadID = parsePayloadID(encoded);

    if (payloadID == 1) {
        transfer = parseTransfer(encoded);
    } else if (payloadID == 3) {
        BridgeStructs.TransferWithPayload memory t =
        parseTransferWithPayload(encoded);
        transfer.payloadID = 3;
        transfer.amount = t.amount;
        transfer.tokenAddress = t.tokenAddress;
        transfer.tokenChain = t.tokenChain;
        transfer.to = t.to;
        transfer.toChain = t.toChain;
        // Type 3 payloads don't have fees.
        transfer.arbiterFee = 0;
        transfer.serviceFee = t.serviceFee;
    } else if(payloadID == 4) {
        transfer = parseTransferReturn(encoded);
    } else {
        revert("Invalid payload id");
    }
}

```


2. Improper positioning of user input validation.

ID: ISKRA-BRIDGE-02

Severity: Tips

Type: Input Validation

Difficulty: N/A

File: bridge/wormhole/bridge/Bridge.sol

Impact

When transferring ERC20 tokens on a cross-chain using Iskra Bridge, user funds may be frozen due to the difference in validation between fromChain and toChain.

Description

Iskra Bridge works in the following order when transferring ERC20 tokens between cross-chains.

- **(Create Request):** A user who wants to transfer a token calls the *transferTokens*(or *transferTokensWithPayload*) function of the Iskra Bridge contract in fromChain.
Input: transfer token address, transfer token amount, recipientChain, recipient address, fee
- fromChain's Iskra Bridge contract locks (or burns) the transfer token, then creates a transfer message and calls *publishMessage* of the wormhole core.
- Guardians who detect events in the Wormhole Core generate VAA when an event is detected.
- **(Claim):** The relayer or user (recipient) passes the VAA value created by guardian to the *completeTransfer*(or *completeTransferWithPayload*) function of the Iskra Bridge contract in toChain.
- toChain's Iskra Bridge contract parses the VAA to verify that the guardians have signed it correctly and parses the transfer message contained in the VAA to mint (or transfer) the transfer token.

While fromChain's *transferTokens* function does not validate that the recipient address value is an EVM address, toChain's *completeTransfer* function validates that the recipient address value is an EVM address.

```
// Execute a Transfer message
function _completeTransfer(bytes memory encodedVm) internal returns (bytes
memory) {
    (IWormhole.VM memory vm, BridgeStructs.Transfer memory transfer) =
    _verifyTransferVM(encodedVm);

    // payload 3 must be redeemed by the designated proxy contract
    address transferRecipient = truncateAddress(transfer.to);
```

<https://github.com/iskraworld/iskra-contracts/blob/v1.2.0-bridgealpha.3/contracts/bridge/wormhole/bridge/Bridge.sol#L434-L439>

The code that validates whether it is an EVM address compares whether the first 12 bytes of bytes32 format are 0, as shown below. Because the EVM address uses only 20 bytes, the first 12 bytes are padded with zeros.

```
/*
 * @dev Truncate a 32 byte array to a 20 byte address.
 *      Reverts if the array contains non-0 bytes in the first 12 bytes.
 *
 * @param bytes32 bytes The 32 byte array to be converted.
 */
function truncateAddress(bytes32 b) internal pure returns (address) {
    require(bytes12(b) == 0, "invalid EVM address");
    return address(uint160(uint256(b)));
}
```

<https://github.com/iskraworld/iskra-contracts/blob/v1.2.0-bridgealpha.3/contracts/bridge/wormhole/bridge/BridgeGetters.sol#L94-L103>

So, Iskra Bridge users can request *transferTokens* by entering the recipient address value as an incorrect address instead of the EVM address when transferring ERC20 tokens between cross-chains, but claiming it is not possible with the *completeTransfer* function. If the user uses the "12 bytes(padding with not null byte) + 20 bytes(address)" instead of the "12 bytes(padding with null byte) + 20 bytes(address)" as the recipient address format, the user's funds will be frozen.

Recommendation

Chains supported by Iskra Bridge are implemented based on EVM. Since the addresses used in the transfer process are always EVM addresses, the *transferTokens* and *transferTokensWithPayload* functions must also verify that the recipient address is an EVM address.

3. gas optimization tips

ID: ISKRA-BRIDGE-03

Severity: Tips

Type: Gas Optimization

Difficulty: N/A

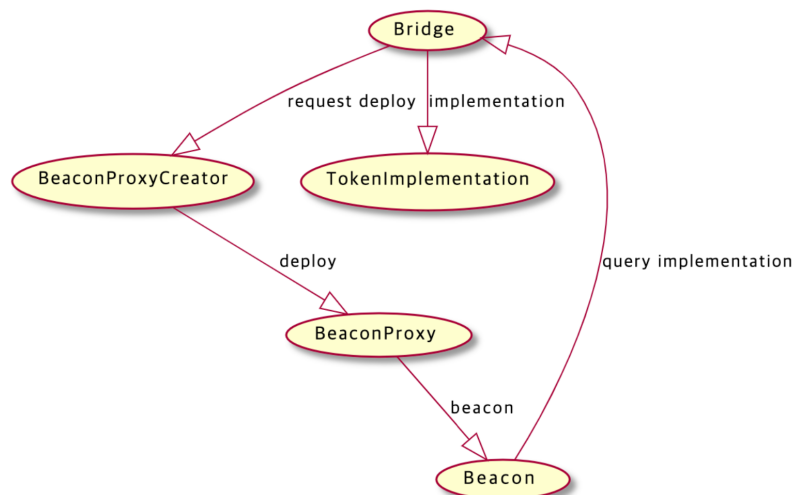
File: bridge/beacon/Beacon.sol

Description

Iskra Bridge uses BeaconProxy when deploying WrappedToken and TokenAdapter.

When you get the implementation address from the BeaconProxy contract, you call the `implementation()` function of the Beacon contract.

The Iskra Team implemented to store the implementation address for the WrappedToken and TokenAdapter in the Bridge contract and then retrieve the WrappedToken address and TokenAdapter address stored in the Bridge contract in the `implementation()` function of the Beacon contract.



Beacon structure

Beacon Contract implements the below code to retrieve implementation address stored in Bridge Contract.

```
contract Beacon is IBeacon {  
    address public provider;
```

```

bytes4 public getterSelector;

constructor(address _provider, bytes4 _getterSelector) {
    provider = _provider;
    getterSelector = _getterSelector;
}

function implementation() external view override returns (address) {
    (, bytes memory raw) =
provider.staticcall(abi.encodeWithSelector(getterSelector));
    return abi.decode(raw, (address));
}
}

```

<https://github.com/iskraworld/iskra-contracts/blob/v1.2.0-bridgealpha.3/contracts/bridge/beacon/Beacon.sol#L8-L21>

When deploying the Beacon contract, it receives the provider value, which is the address of the Bridge contract, and the getterSelector value, which is the 4-byte signature of the function to look up the implementation address.

when the *implementation()* function is called, the implementation address is retrieved from the Bridge contract using the stored provider and getterSelector values.

- Tip1 - immutable

The provider and getterSelector variables are the values set in the contract's constructor and are variables whose values do not change. In the case of immutable values specified in the constructor, using the immutable keyword supported by Solidity is the variable to be directly inserted into the runtime code instead of stored in storage.

As a result, gas costs can be saved. (Reference - 1)

- Tip2 - public/private variable

The provider and getterSelector variables are set to public, but are not used by other contracts.

If a variable is made public, an internal getter function is automatically created at compile time, and branch logic for the getter function is added in the main function bytecode.

As a result, more gas is consumed each time the contract is called.

```

contract Contract {
    function main() {
        memory[0x40:0x60] = 0x80;
        var var0 = msg.value;

        if (var0) { revert(memory[0x00:0x00]); }

        if (msg.data.length < 0x04) { revert(memory[0x00:0x00]); }

        var0 = msg.data[0x00:0x20] >> 0xe0;

        if (var0 == 0x085d4883) {
            // Dispatch table entry for provider()
            var var1 = 0x004e;
            var var2 = func_00A0();
            var temp0 = var2;
            var2 = 0x005b;
            var var3 = temp0;
            var var4 = memory[0x40:0x60];
            var2 = func_0297(var3, var4);
            var temp1 = memory[0x40:0x60];
            return memory[temp1:temp1 + var2 - temp1];
        } else if (var0 == 0x5c60dalb) {
            // Dispatch table entry for implementation()
            var1 = 0x006c;
            var1 = func_00C4();
        }
    }
}

```

<https://ethervm.io> - public variable decompile result

Recommendation

Below is an example of a Beacon contract with gas optimization applied.

I used the immutable keyword and changed the variable to private.

```

contract Beacon is IBeacon {
    address immutable provider;
    bytes4 immutable getterSelector;

    constructor(address _provider, bytes4 _getterSelector) {
        provider = _provider;
        getterSelector = _getterSelector;
    }

    function implementation() external view override returns (address) {
        (, bytes memory raw) =
        provider.staticcall(abi.encodeWithSelector(getterSelector));
        return abi.decode(raw, (address));
    }
}

```

example patch

Reference

1. <https://blog.soliditylang.org/2020/05/13/immutable-keyword/>

Fix

Last Update: 2022.11.11.

#ID	Title	Type	Severity	Difficulty	Status
1	When moving assets between chains, user can avoid paying serviceFee by calling returnTransfer twice	Logic Bug	Medium	Low	Fixed
2	Improper positioning of user input validation	Input Validation	Tips	N/A	Fixed
3	Gas Optimization TIPS	Gas Optimization	Tips	N/A	Acknowledged

Fix Comment

Issue 1 was fixed [in this commit](#).

Issue 2 was fixed [in this commit](#).

Issue 3 was acknowledged by the team. If the contract uses immutable variables, some Klaytn Explorer services have problems with contract verification.

DISCLAIMER

This report does not guarantee investment advice, the suitability of the business models, and codes that are secure without bugs. This report shall only be used to discuss known technical issues. Other than the issues described in this report, undiscovered issues may exist such as defects on the mainnet. In order to write secure smart contracts, correction of discovered problems and sufficient testing thereof are required.

End of Document