QuillAudits

Audit Report, February, 2024

For

taiko

Table of Content

Table of Content

Executive Summary

Project Name

Taiko

Overview

This protocol contains a set of smart contracts. The Bridge contract provides functionality for processing the message/transactions, retrying the message and to recall the message. ERC20Vault, ERC721Vault, ERC1155Vault provides functionality for initiating the calls for sending the tokens, retrying and recalling the message for sending tokens. Also vaults are used to store the tokens and to create bridged tokens. Specifically ERC20Vault provides functionality for changing the bridged token address.

Timeline

4 December 2023 - 27th February 2024

Method

Manual Review, Functional Testing, Automated Testing, etc. All the raised flags were manually reviewed and re-tested to identify any false positives.

Audit Scope

The scope of this audit was to analyze the Taiko codebase for quality, security, and correctness.

Source code

https://github.com/taikoxyz/taiko-mono/tree/based_contestable_zkrollup

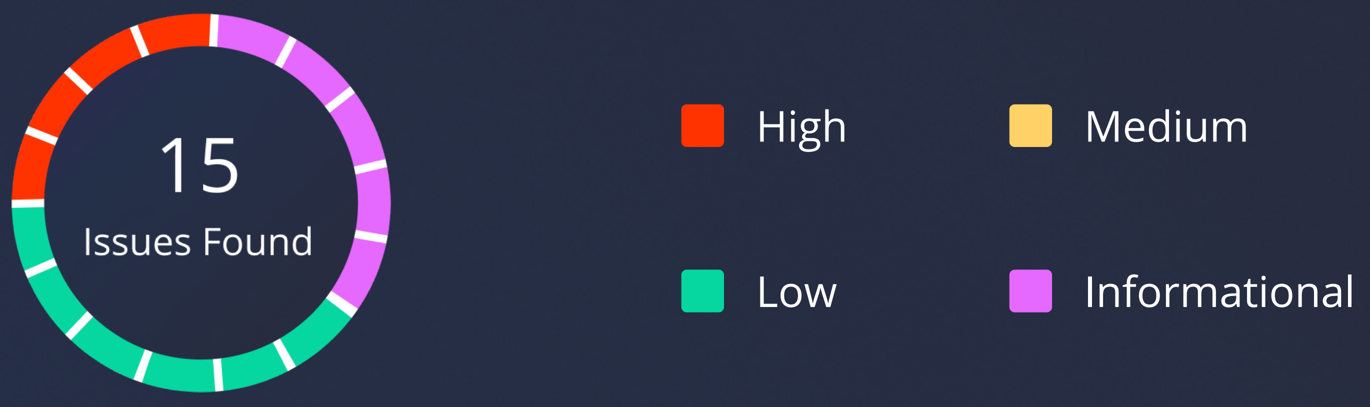
Branch

based_contestable_zkrollup

Fixed in

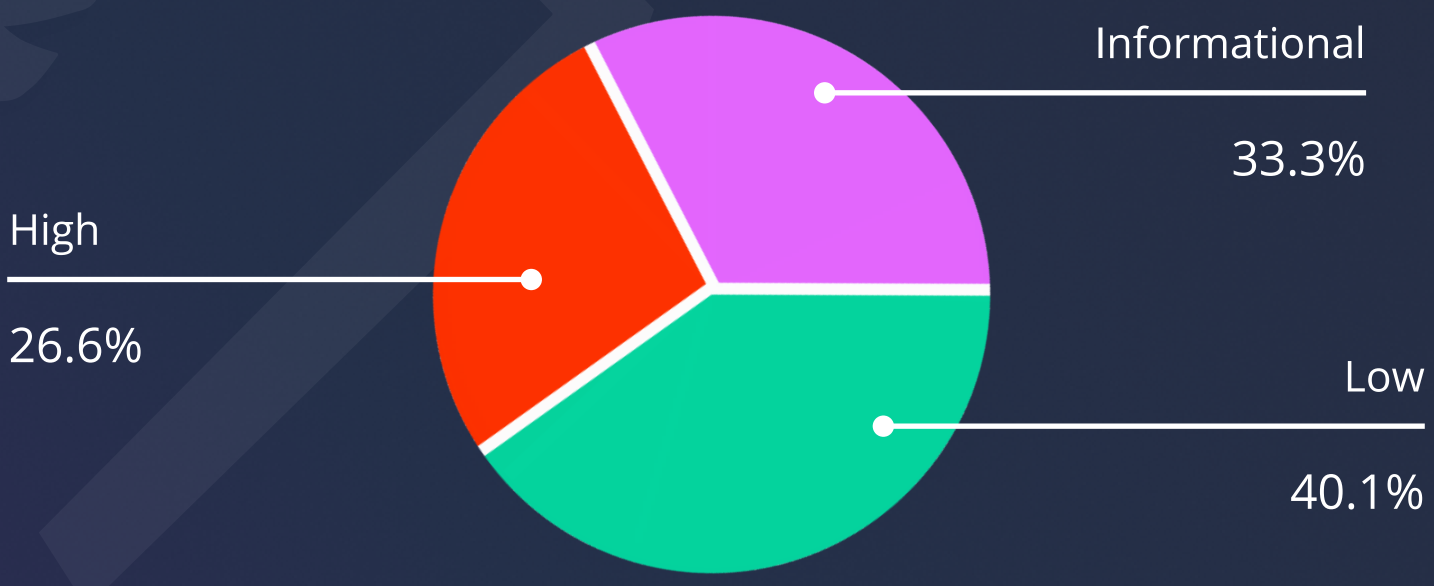
main - f7a12b8601937eef97068c3029c91dff431c03a8 (while most of the issues are fixed in main branch - f7a12b8601937eef97068c3029c91dff431c03a8, There are additional features/improvements added in the main as the Taiko source code is under constant improvement, we have mentioned commit link for some fixes for the specific scope below for possible issue numbers)

Number of Issues per Severity



	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	0	2
Partially Resolved Issues	0	0	0	1
Resolved Issues	4	0	6	2

Security Chart



Checked Vulnerabilities

We scanned the application for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Gas Limit and Loops
- ✓ DoS with Block Gas Limit
- ✓ Use of tx.origin
- ✓ Exception disorder
- ✓ Gasless send
- ✓ Balance equality
- ✓ Byte array
- ✓ Transfer forwards all gas
- ✓ Transaction-Ordering Dependence
- ✓ Redundant fallback function
- ✓ ERC20 API violation
- ✓ ERC721 API violation
- ✓ Malicious libraries
- ✓ Compiler version not fixed
- ✓ Send instead of transfer
- ✓ Style guide violation
- ✓ Unchecked external call
- ✓ Unchecked math
- ✓ Unsafe type inference
- ✓ Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments, match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Hardhat, Foundry.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Issues Found

High Severity Issues

1. Unverified arbitrary data call allows attackers to bypass all validations and pass malicious message hashes directly to the signal service

Files

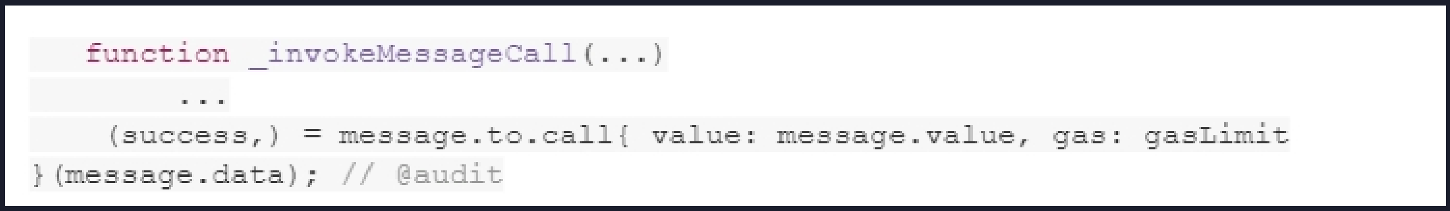
Bridge.sol#L358, SignalService

Function

_invokeMessageCall()

Description

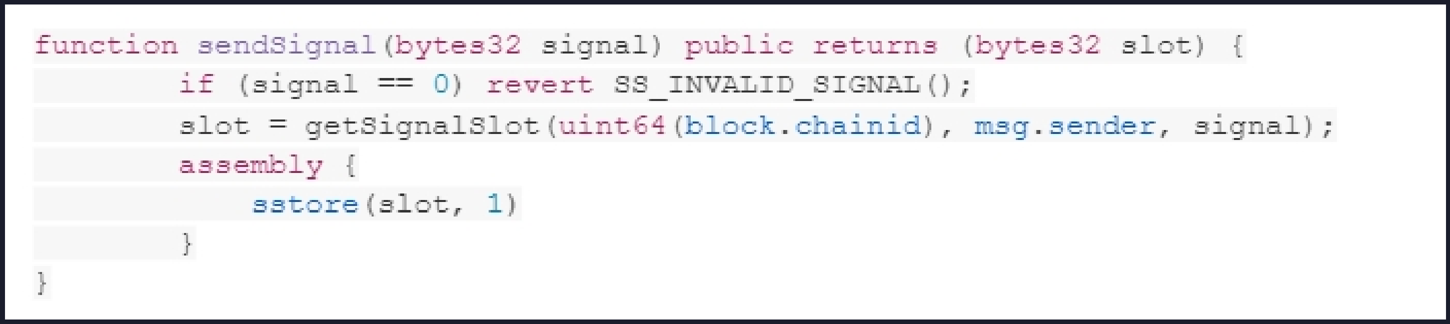
In the _invokeMessageCall() there is a low level external call to the address to specified along with encoded data, it is assumed not to be an issue if originated from the vault contracts, however, it is also intended that message can also be sent directly with the bridge contract in which the to is specified by the user same as the data.



```
    function _invokeMessageCall(...)
          ...
     (success,) = message.to.call{ value: message.value, gas: gasLimit
}(message.data); // @audit
```

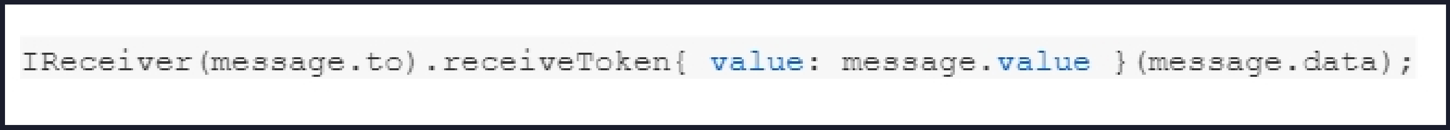
This allows an attacker to make an external call directly to the signal service passing along malicious message hash.

Although users making a call to the signal service directly isn't supposed to be an issue because the address of the msg.sender is attached



```
function sendSignal(bytes32 signal) public returns (bytes32 slot) {
        if (signal == 0) revert SS_INVALID_SIGNAL();
        slot = getSignalSlot(uint64(block.chainid), msg.sender, signal);
        assembly {
            sstore(slot, 1)
        }
}
```

and checked upon proving. However, in the case of this low level call, the msg.sender would be the bridge contract and therefore make the signal valid when proving.



```
IReceiver(message.to).receiveToken{ value: message.value }(message.data);
```

Recommendation

Consider having a receive callback interface function for this contracts to implement and make external calls to this function instead.

Status

Resolved

2. Unchecked transfer causes user to lose funds allocated for fee

Files

ERC20Vault.sol#L285-L323

Function

sendToken(), _handleMessage()

Description

Whenever user calls ERC20Vault::sendToken passing along msg.value for the fee which is collected if all execution goes through successfully including the transfer of the actual funds to be sent performed in ERC20Vault::_handleMessage(), However ERC20Vault::_handleMessage() ignores return value by t.transferFrom({from:msg.sender,to:address(this),amount:amount}). This means whenever transfer fails and returns false, execution won't fail and return but still take user msg.value (for fee) and sends the message for _balanceChange=0.

Recommendation

Use SafeERC wrapper functions for transferring tokens.

Status

Resolved

3. Users would permanently lose their funds if message failed due to incorrect decoding of message.data

Files

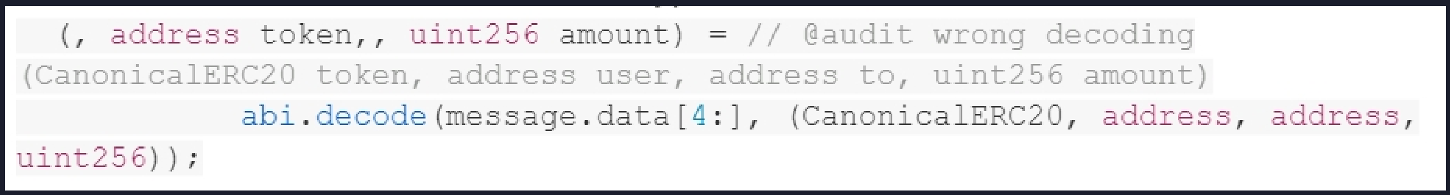
ERC20Vault.sol#L253

Function

onMessageRecalled()

Description

In the onMessageRecalled of the ERC20Vault which is used to retrieve a user's fund back on source chain if status failed, the data which is encoded from the vault contract in the format: (CanonicalERC20 token, address user, address to, uint256 amount), is decoded back to get the amount and the token that was being sent, however this decoding is done incorrectly in the sense that the address that was taken to be the token is actually the address of the user as can be seen from the snippet, the actual token is the first value with the type CanonicalERC20 after the function selector.

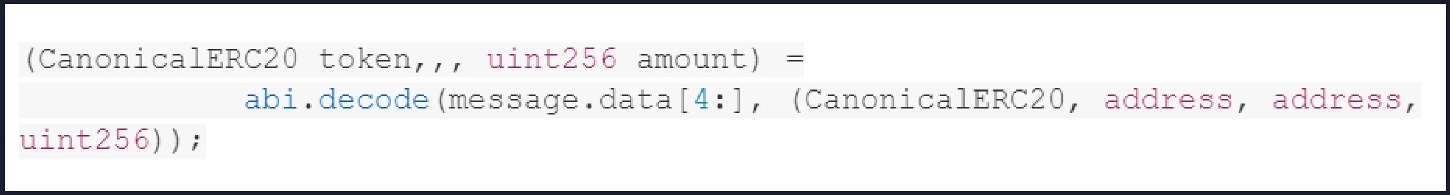


```
  (, address token,, uint256 amount) = // @audit wrong decoding
(CanonicalERC20 token, address user, address to, uint256 amount)
            abi.decode(message.data[4:], (CanonicalERC20, address, address,
uint256));
```

This mistake would cause the function to revert whenever users then attempts to retrieve their funds back.

Recommendation

Make the required changes and adjust the function code accordingly. E.g:



```
(CanonicalERC20 token,,, uint256 amount) =
            abi.decode(message.data[4:], (CanonicalERC20, address, address,
uint256));
```

Status

Resolved

4. Unable to recall the bridged tokens

Files

ERC20Vault.sol#L265, ERC721Vault.sol#161, ERC1155Vault.sol#179

Function

onMessageRecalled()

Description

In the case of recalling a canonical token present on chain A that user wanted to send on different chain (Chain B) , The bridge calls onMessageRecalled() which checks that if (bridgedToCanonical[ctoken.addr].addr != address(0)) is true then mints the token in the if{} block. if false then transfers the token(s). So in this case it will check the value of ctoken address in the bridgedToCanonical mapping. It would be 0 so the if condition will fail and the else{} will be executed as intended which will transfer the previously sent token back to message.owner.

But in the case where user recalls the message which was about sending the bridged token from the chain B to chain A the same onMessageRecalled() function gets executed on the vault by Bridge contract, Now this time it checks the same condition i.e if(bridgedToCanonical[ctoken.addr].addr != address(0)) then mints, otherwise transfers the token to the message.owner . But because this time it's trying to recall the bridged token that got burned while sending and the ctoken (i.e ctoken.addr) in this case would be canonical token address, which is getting encoded in _handleMessage() on src chain vault while sending the token.

So checking if(bridgedToCanonical[ctoken.addr].addr != address(0)) on src chain (the chain where bridged token is deployed in this case) where message is getting recalled will fail because theres no token mapped for bridgedToCanonical[canonical token address]. (The same issue is present in ERC20Vault, ERC721Vault, ERC1155Vault)

Recommendation

The canonicalToBridged mapping should be used to get the bridged token address which can be then minted as a process of recalling the message. Additionally the chain id can be used to check that the token that is getting recalled is canonical token where the transfer needs to be used.

Status

Resolved

Low Severity Issues

1. Using ERC721::_mint() and unsafe ERC721 transfer operations can be dangerous, all leading to permanent loss of NFT

Files

ERC721Vault.sol, ERC721Vault.sol, BridgedERC1155.sol

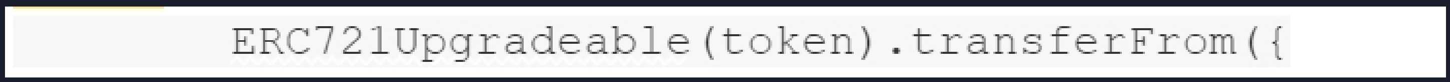
Function

transferFrom(), _mint()

Description

Should revert if _to doesn't support ERC721 by using safeTransferFrom, However it doesn't do that and still sends the token regardless

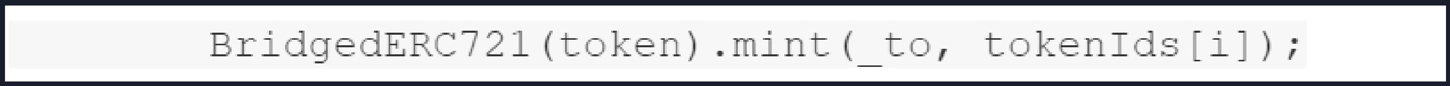
Found in contracts/tokenvault/ERC721Vault.sol Line: 103



```
            ERC721Upgradeable(token).transferFrom({
```

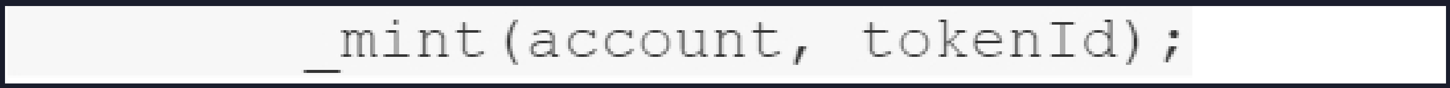
Also using ERC721::_mint() can mint ERC721 tokens to addresses which don't support ERC721 tokens. Use _safeMint() instead of _mint() for ERC721.

Found in contracts/tokenvault/ERC721Vault.sol Line: 112



```
            BridgedERC721(token).mint(_to, tokenIds[i]);
```

Found in contracts/tokenvault/BridgedERC721.sol Line: 66



```
        _mint(account, tokenId);
```

Recommendation

Consider using safeTransferFrom() and safeMint() instead of transferFrom() and _mint().

Status

Resolved

2. OwnableUpgradable uses single-step ownership transfer

Path

OwnerUUPSUpgradable.sol

Function

sendToken(), _handleMessage()

Description

Using single step ownership transfer is not secure, the existing owner can assign any new address as owner, and lose the ownership if the assigned address is incorrect address which can’t perform actions.
Ownable2StepUpgradable provides added safety where the pending owner can accept the ownership due to its two-step process to transfer the ownership.

Recommendation

Consider using Ownable2StepUpgradable instead of OwnableUpgradable.

Status

Resolved

Fixed In

https://github.com/taikoxyz/taiko-mono/pull/16029

3. Already defined batch methods can be used from token contract instead of writing new loops for transferring and minting tokens

Files

ERC1155Vault.sol

Function

transferFrom(), _mint()

Description

receiveToken, onMessageRecalled and _handleMessage functions in the contract use a for loop to call safeTransferFrom and mint methods to transfer and mint tokens.

Instead of using safeTransferFrom methods in the loop, safeBatchTransferFrom can be used to transfer the batch of tokens. And instead of using mint() on the bridged token every time in the loop, the _mintBatch() can be used from the OZ ERC1155 implementation which can be exposed publicly to ERC1155Vault as it's happening in the case of mint().

This should be noted that _mintBatch() and safeBatchTransferFrom() will also use the loops.

Recommendation

Consider using safeBatchTransferFrom and batch minting instead of using safeTransferFrom and minting in the loop.

Status

Resolved

Fixed In

https://github.com/taikoxyz/taiko-mono/pull/16030/files

4. Use encodeCall instead of encodeWithSelector

Files

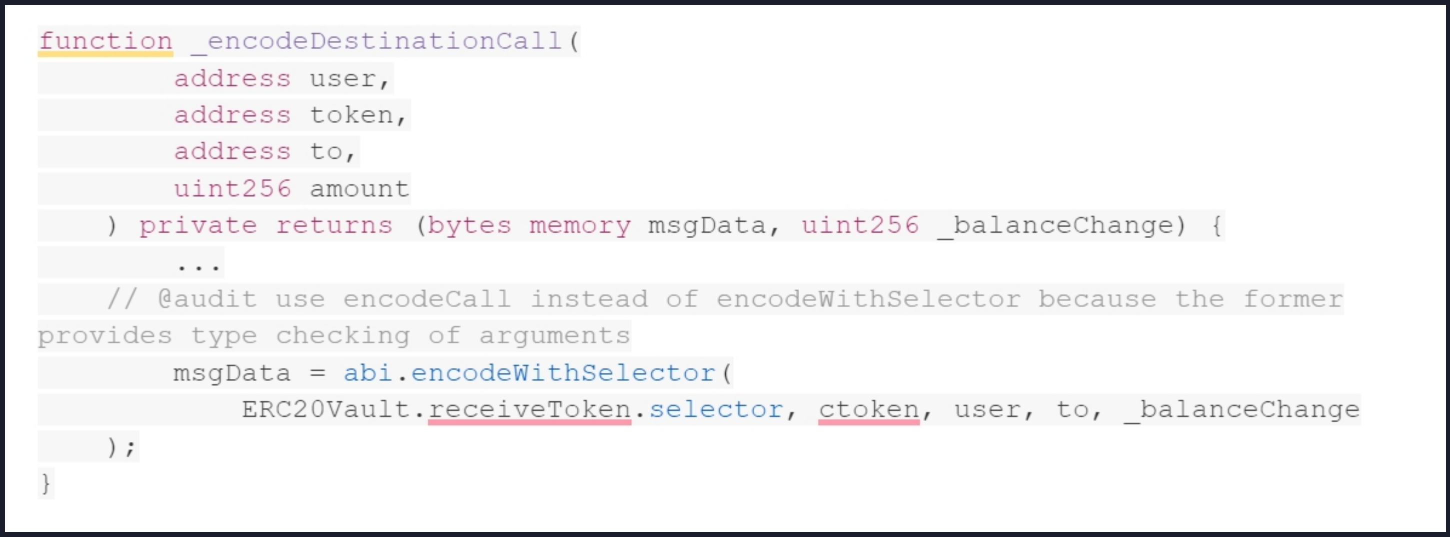
ERC20Vault.sol ERC721Vault.sol ERC1155Vault.sol

Function

_handleMessage()

Description

_encodeDestinationCall (now _handleMessage) private function uses abi.encodeWithSelector to encode the receiveToken function call. This method doesn't provide type checking of function arguments.



```
function _encodeDestinationCall(
        address user,
        address token,
        address to,
        uint256 amount
    ) private returns (bytes memory msgData, uint256 _balanceChange) {
        ...
    // @audit use encodeCall instead of encodeWithSelector because the former
provides type checking of arguments
        msgData = abi.encodeWithSelector(
            ERC20Vault.receiveToken.selector, ctoken, user, to, _balanceChange
    );
}
```

abi.encodeCall will check the arguments type and gives error if not same.

Recommendation

Consider using encodeCall to generate message data for receiveToken function.
Status: Resolved

Status

Resolved

5. Limit the function executed by the processMessage()

Files

Bridge.sol

Function

processMessage()

Description

processMessage allows anyone to execute the transactions on the bridge. There can be multiple possibilities where a user will try to use the transaction initiated from bridge (by call()) to make a call on the arbitrary addresses.

One example of it is a malicious user will try to steal the approvals given to the bridge where a malicious user will execute the transaction on message.to (token address) and will call transferFrom() to transfer it to the arbitrary address.

The action of sending a transaction by bridge can be limited to only certain functions depending on the intended logic similarly what is happening in the recallMessage() for IRecallableSender(message.from).onMessageRecalled call.

Recommendation

Add limitation to the call processMessage() is making to reduce the attack surface.

Status

Resolved

Fixed In

https://github.com/taikoxyz/taiko-mono/pull/15996

6. Possibility of executing the transactions on in-protocol contracts

Files

Bridge.sol

Function

processMessage()

Description

Anyone can execute the the transaction on any arbitrary address with processMessage it is possible that the message.to can be the address used by protocol in the future which only checks that the caller/msg.sender is Bridge contract address for authorization logic, in this case it is possible for anyone to execute the transaction on the in-protocol contract address using processMessage().

Recommendation

The check can be added to restrict calls on any specific contract.

Status

Resolved

Informational Issues

1. Remove unused errors and events

Files

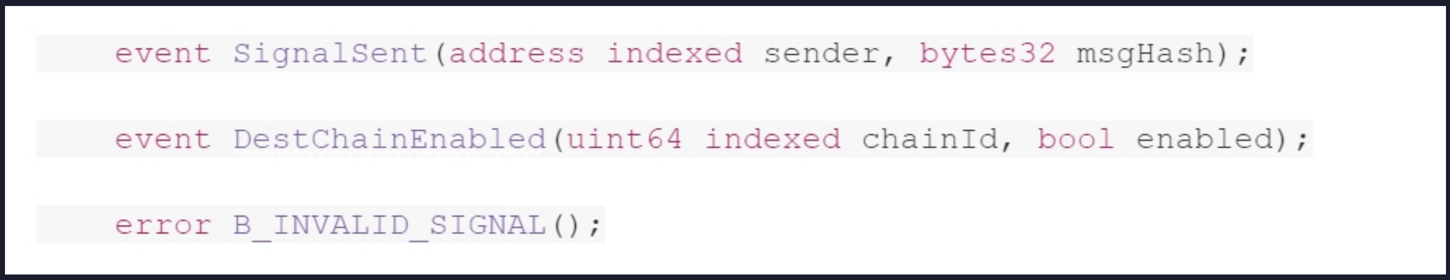
Bridge.sol

Function

processMessage()

Description

Bridge contract contains an unused error B_INVALID_SIGNAL and two unused events SignalSent, DestChainEnabled.



```
    event SignalSent(address indexed sender, bytes32 msgHash);

    event DestChainEnabled(uint64 indexed chainId, bool enabled);

    error B_INVALID_SIGNAL();
```

Recommendation

Consider removing unused error and events.

Status

Resolved

2. Check oldAddress and newAddress are not same

Path

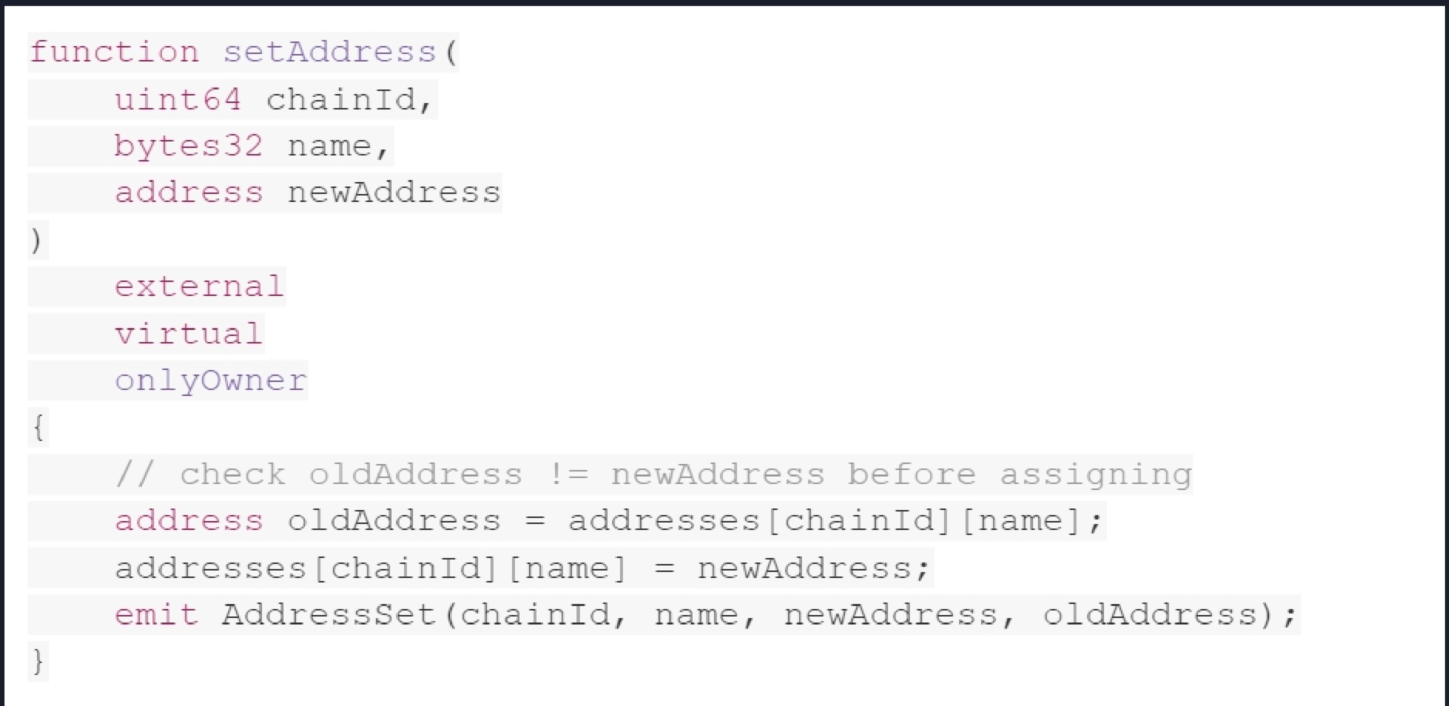
AddressManager.sol

Function

setAddress()

Description

While setting an address the old address and new address can be same. To avoid spending unnecessary gas and emitting a new event for this, having a check before assignment will be a best practice.



```
function setAddress(
    uint64 chainId,
    bytes32 name,
    address newAddress
)
    external
    virtual
    onlyOwner
{
    // check oldAddress != newAddress before assigning
    address oldAddress = addresses[chainId][name];
    addresses[chainId][name] = newAddress;
    emit AddressSet(chainId, name, newAddress, oldAddress);
}
```

Recommendation

Consider checking the newAddress before setting it.

Status

Resolved

Fixed In

https://github.com/taikoxyz/taiko-mono/pull/16031

3. Replace assert with require

Path

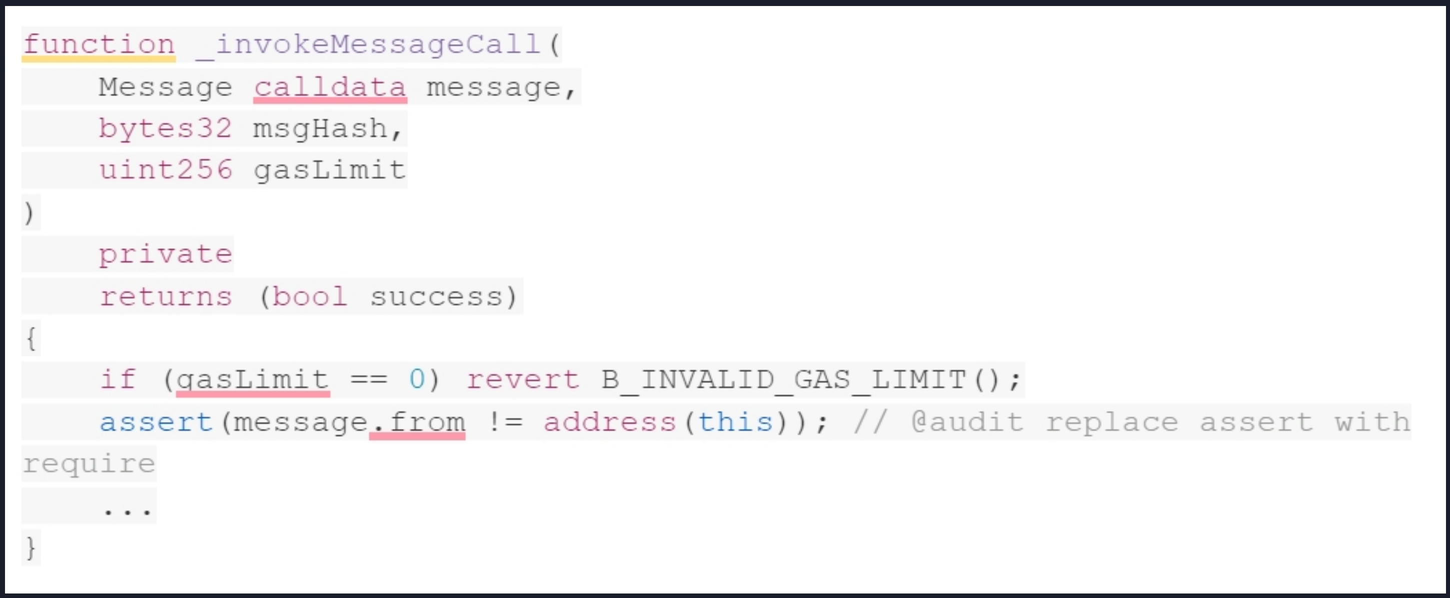
Bridge.sol

Function

transferFrom(), _mint()

Description

_invokeMessageCall function in the contract uses assert to verify a condition, but this should not be used in deployment code as it is not a best practice to do.



```
function _invokeMessageCall(
    Message calldata message,
    bytes32 msgHash,
    uint256 gasLimit
)
    private
    returns (bool success)
{
    if (gasLimit == 0) revert B_INVALID_GAS_LIMIT();
    assert(message.from != address(this)); // @audit replace assert with
require
    ...
}
```

Recommendation

Consider replacing the assert statements with require statements in the contract.

Status

Acknowledged

4. Misleading comments

Path

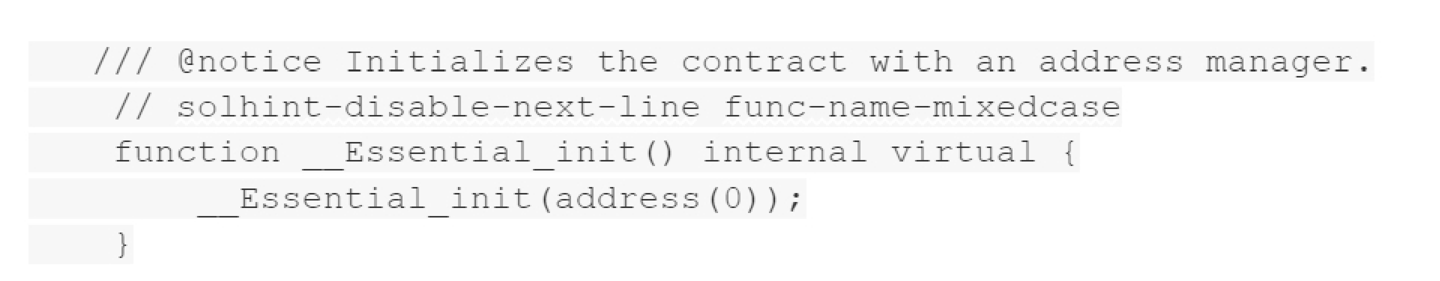
EssentialContract.sol, LibAddress.sol, Bridge.sol#L226-227

Function

-

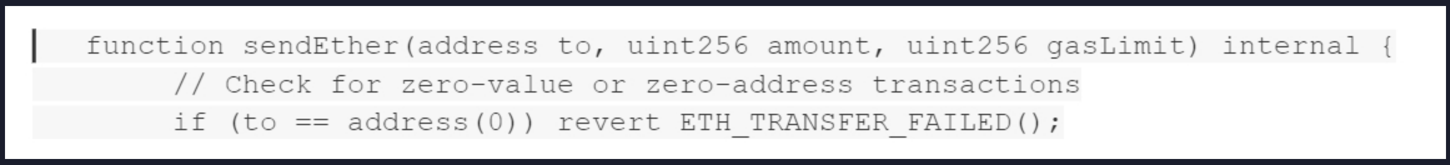
Description

In EssentialContract.sol, __Essential_init() does not take any parameters unlike __Essential_init(address _addressManager) which initializes the AddressResolver contract with an address manager. The comment is incorrect and should be updated to match the code functionality.



```
   /// @notice Initializes the contract with an address manager.
    // solhint-disable-next-line func-name-mixedcase
    function __Essential_init() internal virtual {
        __Essential_init(address(0));
    }
```

In LibAddress.sol, the sendEther function does not check for zero-value transactions as the comment implies. This could be a to-do left unimplemented in the code.



```
    function sendEther(address to, uint256 amount, uint256 gasLimit) internal {
        // Check for zero-value or zero-address transactions
        if (to == address(0)) revert ETH_TRANSFER_FAILED();
```

In Bridge contract on L226-227 a comment states “Use the specified message gas limit if called by the owner, else use remaining gas” but in reality the specified gas in the message should be used if not called by the owner otherwise the remaining gas should be used if the caller is the owner.

Recommendation

Update comments and or code as suggested to match code functionality.

Status

Partially Resolved

5. Check __gaps have configure same size in all contracts

Path

Bridge.sol, ERC20Vault.sol

Function

transferFrom(), _mint()

Description

__gap variable size is not equal to 50 in some contracts. Standard size (50) should be followed, having different sizes isn't logical it is more difficult to maintain the code.

Recommendation

Consider changing the values of __gap variable array.

Status

Acknowledged

Functional Tests

Some of the tests performed are mentioned below:

Bridge:

- ✓ Users can send the message to send the native token.
- ✓ Users can retry the retriable message.
- ✓ Users can recall the failed message.
- ✓ Should revert while processing the message on the chain different than message.destChainId.
- ✓ Should revert while retrying the message on the chain different than message.destChainId.
- ✓ Should revert while recalling the message on the chain different than message.srcChainId.
- ✓ Should revert if signal verification fails on signal service.
- ✓ processing, retrying and recalling messages updates the status of messages.
- ✓ Should not be able to execute the transaction on Bridge.
- ✓ Should not be able to execute the transaction on signal service.

ERC20Vault:

- ✓ Should be able to send canonical tokens on other chain.
- ✓ Should be able to receive the bridge token on other chain.
- ✓ Should be able to get the recalled canonical tokens back.
- ✓ Should be able to send bridge tokens on other chain.
- ✓ Should be able to receive the canonical tokens on other chain.
- ✓ Should be able to get the recalled bridged tokens back.
- ✓ Owner should be able to change the bridge token.

Functional Tests

ERC721Vault:

- ✓ Should be able to send canonical tokens on other chain.
- ✓ Should be able to receive the bridge token on other chain.
- ✓ Should be able to get the recalled canonical tokens back.
- ✓ Should be able to send bridge tokens on other chain.
- ✓ Should be able to receive the canonical tokens on other chain.
- ✓ Should be able to get the recalled bridged tokens back.

ERC1155Vault:

- ✓ Should be able to send canonical tokens on another chain.
- ✓ Should be able to receive the bridge token on other chain.
- ✓ Should be able to get the recalled canonical tokens back.
- ✓ Should be able to send bridge tokens on other chain.
- ✓ Should be able to receive the canonical tokens on other chain.
- ✓ Should be able to get the recalled bridged tokens back.

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the Taiko codebase. We performed our audit according to the procedure described above.

Some issues of High, Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.In The End,Taiko Team resolved almost all issues.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in **Taiko** smart contracts. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of Taiko smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of the **Taiko** to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

850+
Audits Completed

$30B
Secured

800K
Lines of Code Audited

Follow Our Journey

Audit Report
February, 2024

For

taiko

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com