KAWACH

Security Assessment Report

Router Protocol

Version: Final

Date: 8 May 2023

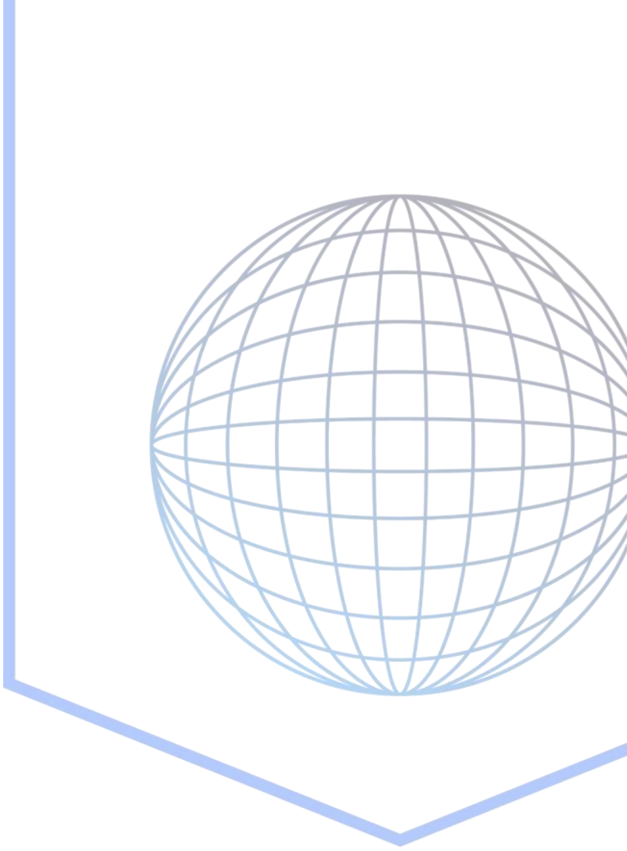


Table of Contents

Licence

Disclaimer

This audit report is based on the findings of the audit conducted by our team, which focused on analyzing the Solidity smart contracts of Router Protocol's gateway contracts.While we have taken all reasonable steps to ensure the accuracy and completeness of the report, we cannot guarantee that our findings are free from errors or omissions. The report should be used for informational purposes only and should not be construed as providing legal or investment advice. We have categorized our findings by level of severity, and provided recommendations to address the security issues we identified. It is our hope that this report will help inform decision making for the development team, and ultimately result in an improved level of security for the contract.

It should be noted that the audit was completely focused on the solidity smart contracts. The project includes some off-chain components which we have assumed to work as per the documentation provided to us.

Introduction

Purpose of this report

Kawach has been engaged by **Router Protocol** to perform a security audit of its contracts for the concentrated liquidity exchange.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behaviour.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebases Submitted for the Audit

The audit has been performed on the following GitHub repositories:

Repository : github.com/router-protocol/router-gateway-contracts/commits/kawach-audit-bug-fix

Date	Version	Commit hash
20/04/2023	Initial Codebase	4f3c97b163dfae60cc03736a5e69b79ccfe023ea
08/05/2023	Fixed issues	4dbf0238dc5d9fbbdd47cda4cb294cace5caf383

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service. This does not require a lot of effort in execution.
High	An attacker can successfully execute an attack that clearly results in operational issues for the service. This also includes any value loss of unclaimed funds permanently or temporary.
Medium	The service may be susceptible to an attacker carrying out an unintentional action, which could potentially disrupt its operation. Nonetheless, certain limitations exist that make it difficult for the attack to be successful.
Low	The service may be vulnerable to an attacker executing an unintended action, but the impact of the action is negligible or the likelihood of the attack succeeding is very low and there is no loss of value.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary.

The status of an issue can be one of the following: **Pending**, **Acknowledged**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Overview

Methodology

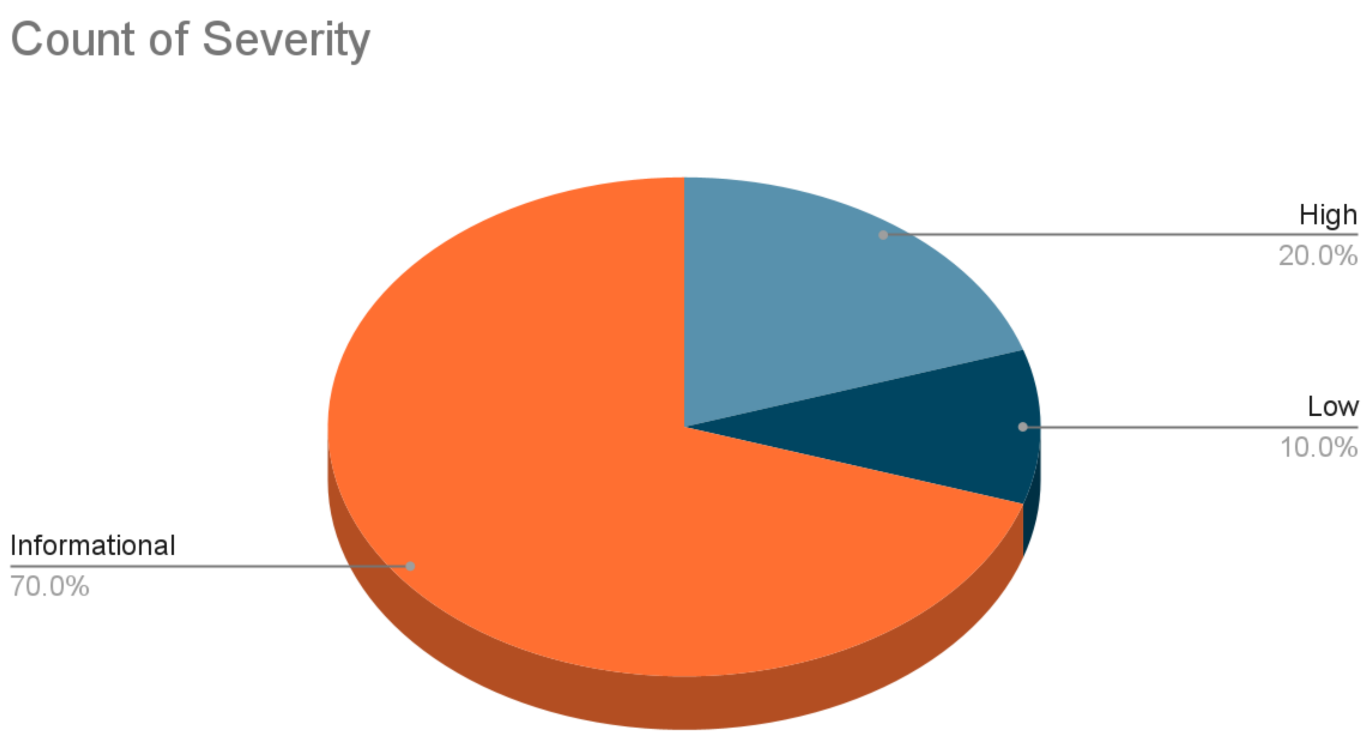
The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line by line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

Router EVM Gateway Contracts are bridge contracts present on each EVM chain that the Router chain is connected with. Using these gateway contracts the Router Chain can interact with all other chains back and forth. Each event that we emit from the gateway contract will be listened to by the router chain orchestrators and they will sign on it and pass it to the router chain. After having ⅔power validation the router chain will process it for further execution.

Summary of Findings



Sr. No.	Description	Severity	Status
1	Signature verification logic can lead to DOS in Gateway	High	Resolved
2	No method to withdraw the collected ETH fees	High	Resolved
3	Acknowledgement reverts if source chain is the same as destination	Low	Resolved
4	Variables can be declared as immutable to save gas	Informational	Resolved
5	Unnecessary Reentrancy guard	Informational	Resolved
6	Redundant v-value check in ECDSA recover	Informational	Resolved
7	Internal functions only called once can be inlined to save gas	Informational	Resolved
8	No need to use the extra memory variable in verifyCrossChainRequest function	Informational	Resolved

Sr. No.	Description	Severity	Status
9	Use capital letter for constant variables	Informational	Resolved
10	Remove unused and redundant code	Informational	Resolved

Detailed Findings

1. Signature verification logic can lead to DOS in Gateway

Severity: High

Description

The `iReceive()` functions validates if the provided Valset arguments and signatures match the payload before executing the handler's function. Inside the `checkValidatorSignatures` function in the `SignatureUtils` library, it iterates over all the signatures and the validator's addresses to verify.

SignatureUtils.sol

```
for (uint64 i = 0; i < _currentValset.validators.length; i++) {
    // If v is set to 0, this signifies that it was not possible to get a signature
    // from this validator and we skip evaluation
    // (In a valid signature, it is either 27 or 28)
    // if (_sigs[i].v != 0) {
    // Check that the current validator has signed off on the hash
    if (!verifySig(_currentValset.validators[i], _theHash, _sigs[i])) {
        revert Utils.InvalidSignature();
    }

    // Sum up cumulative power
    cumulativePower = cumulativePower + _currentValset.powers[i];

    // Break early to avoid wasting gas
    if (cumulativePower > _powerThreshold) {
        break;
    }
    // }
}
```

According to this, if any one of the signatures is found invalid, it reverts. It can lead to a DOS attack, if any of the validators refuses to provide correct signatures. There can be one more scenario of DOS attack where `updateValset` can never be called if any of the current validators refuses to provide valid signatures for the update.

This also means the whole logic of aggregating `cumulativePower` is also not required because the condition `cumulativePower > _powerThreshold` always holds true as it has already been checked in the `ValsetUpdate` library's `validateValsetPower` function.

Recommendation

Instead of reverting on invalid signatures, use `continue` to go to the next iteration and check for the next validator's signatures.

```
for (uint64 i = 0; i < _currentValset.validators.length; i++) {
    // If v is set to 0, this signifies that it was not possible to get a signature
    // from this validator and we skip evaluation
    // (In a valid signature, it is either 27 or 28)
    // if (_sigs[i].v != 0) {
    // Check that the current validator has signed off on the hash
    if (!verifySig(_currentValset.validators[i], _theHash, _sigs[i])) {
        continue;
    }

    // Sum up cumulative power
    cumulativePower = cumulativePower + _currentValset.powers[i];

    // Break early to avoid wasting gas
    if (cumulativePower > _powerThreshold) {
        break;
    }
    // }
}
```

Status

Resolved

Comments : *It is different from the recommended approach because we will never get the wrong signature for the router chain. It might be possible that we will not get the signature for any individual validator, but in that case, we can pass an empty signature, and for this signature validation we will bypass using sig.v == 0 check. In case of wrong or tempered signature added from the relayer side. The relayer will bear the gas cost of the reverted transaction.*

2. No method to withdraw the collected ETH fees

Severity: **High**

Description

There is no function in the contract to withdraw the ETH collected as fees from the contract. The `setDappMetadata`, `iSend` and the `receive` function facilitate ETH transfers to the Gateway contract.

Recommendation

Add a withdraw function with appropriate access control to withdraw ETH from the contract.

Status

Resolved

3. Acknowledgement reverts if source chain is the same as destination

Severity: **Low**

Description

There is no check for `destChainId != chainId` in the `iSend` function. This allows sending and receiving messages on the same chain. But iReceive sets the `nonceExecuted` to true for this chain.

iReceive() [GatewayUpgradeable.sol]

```
nonceExecuted[requestPayload.srcChainId][requestPayload.requestIdentifier] = true;
```

Now the `iAck` function requires that the `nonceExecuted` should equal false for this chain and nonce to proceed with the acknowledgement transaction.

iAck() [GatewayUpgradeable.sol]

```
require(
   nonceExecuted[crossChainAckPayload.destChainId][crossChainAckPayload.ackRequestIdentifier] ==
false,
   "C06"
);
```

Recommendation

Either revert whenever the `destChainId == chainId` or change the nonce logic to handle the cases when receive and acknowledgement transactions are also on the same chain.

Status

Resolved

4. Variables can be declared as immutable to save gas

Severity: Informational

Description

Variables only set in the constructor and never edited afterwards should be marked as immutable to save gas.

AssetVault.sol

```
address public gateway;
ERC20PresetMinterPauser public routeToken;
```

DelayASM.sol

```
address public gatewayContract;
address public owner;
```

Recommendation

Declare these variables as immutables.

Status

Resolved

5. Unnecessary Reentrancy guard

Severity: Informational

Description

There is an unnecessary reentrancy guard on the deposit and withdraw functions. There is no other external call except to the Route token contract.

AssetVault.sol

```
function deposit(uint256 amount, address caller) external nonReentrant {
function handleWithdraw(uint256 amount, address recipient) external nonReentrant {
```

Recommendation

Remove the nonReentrant modifiers

Status

Resolved

6. Redundant v-value check in ECDSA recover

Severity: Informational

Description

The v-value checks in the ECDSA library's recover function are redundant as the precompile already checks if the value is 27 or 28.

ECDSA.sol

```
if (v != 27 && v != 28) revert InvalidV();
```

Recommendation

Use the latest version of ECDSA library from Openzeppelin, as it has some other gas optimizations as well.

Status

Resolved

7. Internal functions only called once can be inlined to save gas

Severity: Informational

Description

Inlining internal/private functions that are only called once can save gas.

DelayASM.sol

```
function isRequestRejected(bytes32 id) internal view returns (bool) {
   return delayedTransfers[id];
}


function verifyCrossChainRequest(
   uint256 eventNonce,
   uint256 requestTimestamp,
   ...


bool isRejected = isRequestRejected(id);
```

Recommendation

To save on gas costs, directly fetch the value from the `delayedTransfers` mapping inside `verifyCrossChainRequest`, instead of using an internal function to do so.

Status

Resolved

8. No need to use the extra memory variable in verifyCrossChainRequest function

Severity: Informational

Description

`isRejected` bool variable is not required. You can directly fetch the value of `delayedTransfers` mapping only once and use it.

Recommendation

You can change the `verifyCrossChainRequest` to the following :

DelayASM.sol

```
function verifyCrossChainRequest(
   uint256 eventNonce,
   uint256 requestTimestamp,
   string calldata srcContractAddress,
   string calldata srcChainId,
   bytes calldata payload,
   address handler
) external view returns (bool) {
   require(msg.sender == gatewayContract, "Caller is not gateway");
   bytes32 id = keccak256(
       abi.encode(eventNonce, requestTimestamp, srcContractAddress, srcChainId, payload, handler)
   );
   if (isRequestRejected(id)) {
       return false;
   }
   _validateDelay(requestTimestamp);
   return true;
}
```

Then access and update the variables as `cache.token0LimitOrderFee` and `cache.token1LimitOrderFee` inside the loop. At the end of the loop again update the storage variables.

Status

Resolved

9. Use capital letter for constant variables

Severity: **Informational**

Description

Constants should be named with all capital letters with underscores separating words.

Utils.sol

```
uint64 constant constantPowerThreshold = 2791728742;
```

Recommendation

Use the variable name `CONSTANT_POWER_THRESHOLD`.

Status

Resolved

10. Remove unused and redundant code

Severity: Informational

Description

There are unused structs and errors which should be removed to improve code readability.

Utils.sol

```
struct Signature {
   uint8 v;
   bytes32 r;
   bytes32 s;
}

error InvalidValsetNonce(uint256 newNonce, uint256 currentNonce);
```

IGateway.sol

```
function crossChainRequestDefaultFee() external view returns (uint256 fees);
```

GatewayUpgradeable.sol

```
modifier isSelf() {
   // "ExecuteHandleCalls : Can only be called by Current Contract" => "C01"
   require(_msgSender() == address(this), "C01");
   _;
}



function toAddress(bytes memory _bytes) internal pure returns (address contractAddress) {
   bytes20 srcTokenAddress;
   assembly {
       srcTokenAddress := mload(add(_bytes, 0x20))
   }
   contractAddress = address(srcTokenAddress);
}

function toBytes(address a) public pure returns (bytes memory b) {
   assembly {
       let m := mload(0x40)
       a := and(a, 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF)
       mstore(add(m, 20), xor(0x140000000000000000000000000000000000000000000, a))
       mstore(0x40, add(m, 52))
```

```
        b := m
    }
}
```

ValsetUpdate.sol

```
import "../SignatureUtils.sol";
```

Recommendation

Remove any unused code from the contracts.

Status

Resolved

Conclusion

Overall, the audit revealed several security issues that had to be addressed to ensure the contract's security. All of the identified bugs have been fixed, with specific changes made to the code to address each issue.