PUBLIC

Code Assessment

of the Circle EVM Bridge Smart Contracts

December 06, 2022

Produced for

CIRCLE

by

CHAINSECURITY

Contents

1 Executive Summary 3

2 Assessment Overview 5

3 Limitations and use of report 11

4 Terminology 12

5 Findings 13

6 Resolved Findings 17

7 Notes 20

1 Executive Summary

Dear Circle Team,

Thank you for trusting us to help the Circle with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Circle EVM Bridge according to Scope to support you in forming an opinion on their security risks.

Circle implements a Cross-Chain Transfer Protocol (CCTP), allowing bridging native tokens from a source chain to a destination chain. The CCTP relies on an off-chain attestation service to sign transfer messages, which is currently operated by Circle.

The most critical subjects covered in our review are signature handling, event handling, access control and functional correctness. Security regarding all the aforementioned subjects is high.

The general subjects covered are trustworthiness, upgradeability, gas efficiency and documentation. The contracts in the scope of this review are not upgradeable, however, several accounts are required to be trusted, see Roles and Trust Model. Also, we have highlighted accounts of high importance in Potential single points of failure. The project has extensive documentation and inline code specification.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	11
• Code Corrected	6
• Code Partially Corrected	2
• Acknowledged	3

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Circle EVM Bridge repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	17 Oct 2022	ff4850e0e152a90bc74309a3dbf6e1bf2fb4fc42	Initial Version
2	05 Dec 2022	8f048e2d8a87d87d7e310c3c424dfa255c4932c9	Version 2
3	06 Dec 2022	40111601620071988e94e39274c8f48d6f406d6d	Version 3

For the solidity smart contracts, the compiler version `0.7.6` was chosen.

The smart contracts inside `src` folder were in the scope of this code assessment.

2.1.1 Excluded from scope

Third-party libraries, tests, and any file not listed above are excluded from the scope of this review. Furthermore, external token contracts and their internal rate limiting mechanism were not in scope. Finally, functions in the library `TypedMemView` that are not used in the codebase are excluded from this code assessment.

2.2 System Overview

This system overview describes the initially received version (**Version 1**) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Circle EVM Bridge is a Cross-Chain Transfer Protocol (CCTP) that allows messages to pass across different chains and enables the bridging of tokens across such chains (The current implementation targets cross-chain USDC). The architecture of the bridge relies on a centralized and trusted entity, namely the Attestation service, which listens for specific events and signs them. Three steps are required for a message to be transferred across the chain:

1. the user calls `sendMessage` in `MessageTransmitter`, which emits a specific event;
2. user should request off-chain signatures from the Attestation service after the event is emitted;
3. user calls `receiveMessage` in the destination chain to broadcast the message.

On a high level, the CCTP protocol is designed in two layers:

- Transmission layer: `MessageTransmitter` exposes functionalities to users to send arbitrary messages to any recipient address in a destination domain (chain).

- Logic layer: `TokenMessenger` sends and receives messages from `MessageTransmitter` and has the responsibility of managing the bridged tokens on the respective domain, i.e., burning and minting accordingly.

The design of the protocol in two layers allows the transmission layer, cross-chain message passing, to be used by other applications and use cases.

2.2.1 Message Transmitter

`MessageTransmitter` contract implements the transmission layer of the protocol and is agnostic to the applications operating at the logic layer. There is only one instance of this contract active at any time in a supported chain. The deployer of the contract sets the `localDomain`, which is of type `uint32` and is unique for each chain.

`MessageTransmitter` provides the following functionalities:

- `sendMessage`: anyone can call this function to send an arbitrary `messageBody` to a `recipient` in a `destinationDomain`. To prevent replay attacks, each message includes a nonce that is checked in the destination domain. If the message is formatted correctly, the event `MessageSent` is emitted.
- `sendMessageWithCaller`: is similar to the `sendMessage` function, but in addition, allows to specify the address that can broadcast the message in the destination domain.
- `receiveMessage`: anyone can call this function to broadcast a message that has been initiated in a remote domain. The caller should present exactly the required threshold of signatures received from the attestation service. The signatures should also be ordered in ascending order based on the signers' addresses. If successful, the function calls `handleReceiveMessage` in the `recipient` address.
- `replaceMessage`: this function allows `msg.sender` to modify a message that has already been sent; however, calling this function is restricted only to the caller in the destination domain. The new message uses the same nonce as the original message. Hence, the first message broadcasted to the destination domain consumes the nonce and invalidates other messages.
- `setMaxMessageBodySize`: can be called only by the `owner` of the contract and sets an upper limit on the size of `messageBody` used by logic layers.
- `updatePauser`: can be called only by the `owner` to set the account that can `pause/unpause` the contract.
- `updateRescuer`: can be called only by the `owner` to set the account that can transfer out any ERC20 token held by the contract.
- `updateAttesterManager`: can be called only by the `owner` to set the account that can add and remove attesters, and set the threshold for required signatures to broadcast a cross-chain message.

The attestation service is composed of a set of centralized servers which listen to `MessageSent` events emitted by `MessageTransmitter` and sign them upon user request.

2.2.2 Token Messenger

`TokenMessenger` contract operates on the logic layer of the protocol. On deployment, it is linked to the correct instance of `MessageTransmitter`. The `TokenMessenger` does not inherit `Pausable`. Hence, the contract cannot be paused. However, if the underlying `MessageTransmitter` is deprecated, this contract becomes nonoperational.

`TokenMessenger` implements the following functionalities:

- `depositForBurn`: anyone can call this function to bridge tokens to another chain. The caller specifies the number of tokens to be bridged and the recipient address in the destination domain. The function burns tokens in the source domain (chain) and composes a `BurnMessage`, which is forwarded to `sendMessage` in `MessageTransmitter`.

- `depositForBurnWithCaller`: is similar to `depositForBurn`, with the only difference being that it specifies the caller address that can broadcast the message in the destination domain.
- `replaceDepositForBurn`: allows `msg.sender` to modify the receiving address of the minted tokens and the caller address in the destination domain for a message already created. This function calls `MessageTransmitter.replaceMessage`; hence, all modified messages use the same nonce and once one message is broadcasted to the destination domain, earlier messages with the same nonce become invalid.
- `handleReceiveMessage`: this function can be called only by `MessageTransmission` and only if the initiator of the cross-chain message is whitelisted as the address of the token messenger in the remote domain. The verification of signatures is performed in the transmission layer (`MessageTransmitter`); hence, `handleReceiveMessage` only checks that the received `messageBody` is a valid `BurnMessage`. If successful, a call to the `localMinter` is made to mint the respective tokens.
- `add/removeRemoteTokenMessenger`: can be called only by the `owner` of the contract to set or remove the address of the respective `TokenMessenger` in a remote domain.
- `add/removeLocalMinter`: can be called only `owner` of the contract to set or remove the address of the `localMinter`. If `localMinter` is set to `address(0)`, `TokenMessenger` cannot process burning or minting operations.
- `updateRescuer`: can be called only by the `owner` to set the account that can transfer out any ERC20 token held by the contract.

2.2.3 Token Minter

The approach used by this token bridge is to burn tokens in one chain and mint them in the other chain. This approach is different from the bridges, which lock tokens on one side and mint wrapped tokens on the other. Therefore, the protocol uses a special contract `TokenMinter` that should have minting permission for supported tokens.

`TokenMessenger` implements the following functionalities:

- `mint`: this function can be called only by the local `TokenMessenger` and mints the respective tokens to the recipient address `to`. Supported tokens, such as `USDC`, can internally restrict the number of tokens allowed to be minted by `TokenMinter` as a rate limiting mechanism in case of emergencies.
- `burn`: this function also can be called only by the local `TokenMessenger` and implements another rate limiting mechanism, which allows burning tokens up to a threshold set by the `tokenController`.
- `add/removeLocalTokenMessenger`: can be called only by the `owner` of the contract to set or remove the address of the local `TokenMessenger` contract.
- `setTokenController`: can be called only by the `owner` of the contract to set a privileged account `tokenController`, which can set the maximum amount of tokens allowed to be burned per call. Importantly, `tokenController` is also trusted to correctly link (and unlink) token addresses in a domain with the respective token addresses in remote domains.
- `updatePauser`: can be called only by the `owner` to set the account that can `pause/unpause` the contract.
- `updateRescuer`: can be called only by the `owner` to set the account that can transfer out any ERC20 token held by the contract.

2.3 Execution Flow

Bridging tokens through this protocol has three main steps: 1) Burning in the local domain; 2) Retrieving signatures off-chain from the Attestation service; 3) Minting tokens in the destination domain.

Burning in the local domain requires the following steps:

1. The user calls `depositForBurn` of the local `TokenMessenger`.
2. `TokenMessenger` calls into the local `TokenMinter` to burn the user's local token.
3. `TokenMessenger` creates a `BurnMessage`, including the destination domain as well as the beneficiary address in this domain.
4. After formatting the `BurnMessage`, `TokenMessenger` calls `sendMessage` in `MessageTransmitter`.
5. `MessageTransmitter` eventually encapsulates this `BurnMessage` as a payload for `Message` and emits an event containing this payload.

The caller in the destination domain should retrieve from the attestation service the required signatures for the event log of interest. After fetching the attestation from a minimum pre-defined number of whitelisted attesters, the signature alongside the original message should be fed into the `MessageTransmitter` in the destination domain:

1. Caller triggers the minting operation in the destination domain by feeding in the signed message to function `receiveMessage` in `MessageTransmitter`.
2. After verifying the signatures, `MessageTransmitter` calls `handleReceiveMessage` of the recipient (the respective `TokenMessenger`).
3. Eventually, `mint` in the token contract is called by `TokenMinter` in the destination domain, which mints tokens with a 1:1 ratio with burned tokens.

2.4 Roles and Trust Model

All contracts presented above have privileged accounts that need to be trusted to behave correctly for the bridge to function as expected. In general, we assume the deployers and accounts with the `owner` role as fully trusted and they configure contracts with the correct parameters. We detail these accounts below and highlight the critical ones in Potential single points of failure.

2.4.1 Message Transmitter

The following accounts are considered fully trusted and should always behave non-maliciously and in the best interests of the protocol:

- `deployer`: sets the immutable variables `localDomain` and `version` of the contract, sets the initial maximum size for the message body, and gets the `owner` and `attesterManager` roles on deployment.
- `owner`: can transfer ownership to another account, update the maximum size for the message body, and set addresses for `rescuer` and `pauser` roles.
- `attesterManager`: manages the on-chain configuration for verifying signatures from the Attestation service. This account can enable or disable attesters at any point, set the threshold for the required signatures, and finally transfer the role to another account.
- `pauser`: can toggle the flag `paused` and if set to `true` prevents the contract from processing cross-chain messages.

The contract has another privileged account `rescuer`, which can transfer out any ERC20 token, that was sent by mistake to the contract. This account does not affect the security of the system; however, we assume it behaves non-maliciously on transferring tokens.

2.4.2 Token Messenger

The following accounts are considered fully trusted and should always behave non-maliciously and in the best interests of the protocol:

- `deployer`: sets the immutable variables `localMessageTransmitter` and `messageBodyVersion`, and gets the `owner` role on deployment.
- `owner`: sets the respective addresses of remote token messenger contracts, can update the address serving as `localMinter` and the `rescuer` address.
- `localMinter`: is responsible for minting and burning tokens in the domains supported by the system. This account should be whitelisted in the supported tokens and have permission to mint new tokens. We assume `localMinter` is a fully trusted smart contract.

This contract also has another privileged account `rescuer`, which can transfer out any ERC20 token, that was sent by mistake to the contract and we assume it behaves non-maliciously when transferring tokens.

2.4.3 Token Minter

The following accounts are considered fully trusted and should always behave non-maliciously and in the best interests of the protocol:

- `owner`: sets and removes address of the local `TokenMessenger` which can trigger `mint` or `burn` calls. `owner` can also set addresses for roles `rescuer`, `pauser` and `tokenController`, and transfer `owner` role to another account.
- `pauser`: can toggle the flag `paused` and if set to `true` does not allow minting or burning tokens.
- `tokenController`: links token in the local domain with the corresponding tokens in the remote domains, and can set the maximum amount that can be burned per function call.

2.4.4 Attestation Service

The whitelisted servers (attesters), that compose the attestation service, are trusted to sign only authentic messages emitted by the `MessageTransmitter` contract. We assume attesters sign messages only for events that have been emitted in finalized blocks.

2.4.5 Upgradeability

The contracts in the scope of this review are not upgradeable; thus, new versions of contracts might be deployed and old ones get deprecated. We assume there is only one version of contracts `MessageTransmitter` and `TokenMessenger` active at the same time in any domain. As versions of contracts are part of the message signed by attesters, we assume future versions of contracts are backward compatible, i.e., users burning tokens with an old version should be able to mint respective tokens in new versions of the contracts. In this report we have only reviewed the first version of the contract, any future version is not in the scope of this review.

2.4.6 Rate limiting mechanisms

The protocol assumes two rate limiting mechanisms. The first is implemented in `tokenMinter` and sets a maximum amount of tokens that can be burned per function call. However, this is not a hard restriction, see Inconsistent natspec descriptions. The second restriction is implemented internally in the token contracts, e.g., minter allowance in USDC token, and is outside the scope of this review.

2.4.7 ERC20 Tokens

Any external token used by the system is considered fully trusted and should be carefully assessed before being whitelisted as a supported token. We assume only ERC20-compliant tokens without special behavior (e.g., inflationary/deflationary tokens, delayed finality, etc.) and implementing the `IMintBurnToken` interface is supported by the system. Finally, supported tokens should also implement the function `burn` which reverts on failure.

2.4.8 Users

Users interacting with the system are assumed to be untrusted. However, we assume users fully control address `msg.sender` calling `TokenMessenger` or `MessageTransmitter`, i.e., others should not be able to call replace functions for a message created by someone else.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	5

- Floating Dependencies Versions Acknowledged
- Gas Optimizations Acknowledged
- Inconsistent Natspec Descriptions Code Partially Corrected
- Missing Sanity Checks Code Partially Corrected
- Potential Event Reordering Due to Reentrancy in MessageTransmitter Acknowledged

5.1 Floating Dependencies Versions

Design Low Version 1 Acknowledged

The versions of the contract libraries imported as git submodules by the foundry are not fixed. With new versions being pushed to the dependency repositories, the imported code can change (e.g., via `forge update`) and lead to unexpected behavior by the smart contracts of the project.

The version of the foundry dependency can be specified as described here.

Acknowledged:

Circle acknowledged this issue and decided to keep the code unchanged due to the following reason:

```
The dependencies in repository are pinned git submodules, which won't be changed
without explicitly committing a new version to master, so no change is needed.
```

We would like to highlight that the pinned version of OpenZeppelin dependency is `4.3.1` which includes a vulnerability in signature handling, however the reviewed code is not affected.

5.2 Gas Optimizations

Design Low Version 1 Acknowledged

1. The MessageTransmitter contract uses a mapping of boolean values to keep track of used nonces, which is inefficient, due to the Solidity compiler automatically padding bool values with zeroes when writing them to storage. It is more efficient to use a mapping of a type such as uint256 that takes up an entire storage slot as the bool values anyway cannot be packed in this case.
2. Functions sendMessage and sendMessageWithCaller declare a return variable _nonce but it remains unused as the variable _nonceReserved is returned in both cases.
3. Attestable contract has a constant value of 65 assigned to immutable variable signatureLength. Changing its type to constant reduces slightly the gas consumption.
4. At the sending end, MessageTransmitter keeps track of available nonces for each destination domain. The contract could be made more efficient in terms of storage used if a single global nonce is used for all remote domains.
5. Function _recoverAttesterSignature computes a hash of _message and then calls recover from ECDSA library to get the address of the signer. This function is only called inside the for-loop in function _verifyAttestationSignatures, therefore causing redundant computation of the hash for the same message.
6. The function disableAttester performs two calls to getNumEnabledAttesters which performs an SLOAD operation. Although the second SLOAD costs less (100 gas) due to storage being warm at that point, the function could be optimized by storing the value in memory.
7. Similarly, the function addLocalTokenMessenger performs an unnecessary SLOAD when emitting the event.
8. The location of the following arguments can be changed from memory to calldata to make them more gas-efficient: messageBody in MessageTransmitter.sendMessage; and newMessageBody in MessageTransmitter.replaceMessage.
9. The function encodeHex in the library TypedMemView always checks if the iterator is not on the 16th byte:

   ```
   for (uint8 i = 31; i > 15; i -= 1) {
       uint8 _byte = uint8(_b >> (i * 8));
       first |= byteHex(_byte);
       if (i != 16) {
           first <<= 16;
       }
   }
   ```

   As an improvement, the loop can iterate in the range i > 16 so the if statement inside the loop can be removed. The same optimization is possible for the next loop which iterates over the lower 16 bytes. By doing so, gas consumption would be decreased.

Acknowledged:

Circle has applied most of the optimizations listed above. More specifically, optimizations 1-6 and 8 were implemented in the updated codebase. Optimizations 7 and 9 were acknowledged but not addressed in code. We detail the fixes:

1. usedNonces is changed to be a mapping of bytes32 to uint256.
2. Circle has corrected both sendMessage and sendMessageWithCaller.
3. signatureLength is changed to be a constant.
4. MessageTransmitter keeps track of the next available nonce via keeping a scalar variable, namely nextAvailableNonce.

5. In `_verifyAttestationSignatures` the digest of the message is firstly calculated and sent down to each call of `_recoverAttesterSignature`.
6. `disableAttester` fetches length of the enabledAttesters and stores it in a memory variable, instead of accessing the storage twice.
7. Circle has acknowledged this optimization but has decided to keep the code unchanged as the function `addLocalTokenMessenger` is not expected to be called often.
8. `messageBody` in `MessageTransmitter.sendMessage`, `originalMessage` and `newMessageBody` in `MessageTransmitter.replaceMessage` are changed to calldata.
9. Circle has decided to keep the `TypedMemView` library as-is.

5.3 Inconsistent Natspec Descriptions

Correctness | Low | Version 1 | Code Partially Corrected

The natspec description of the following functions is not consistent with the implementation:

1. `_sendMessage`: `@dev Increment nonce, ...` is not aligned with the implementation.
2. `_getLocalToken`: `@dev Reverts if unable to find an enabled local token...`, but the implementation does not revert.
3. `onlyWithinBurnLimit`: `... burn limit per-transaction for given 'burnToken'`. The modifier only checks that the limit is not exceeded in a single function call, however, if multiple calls are executed within a transaction, the limit per-transaction is not enforced.
4. `BurnMessage` library: `version` field is declared as 4 bytes, but the type is set to `uint8` instead of `uint32`.
5. To fetch the 12 bytes containing `loc`, a variable of `TypedMemView` should be shifted 120 bits (3 empty + 12 `len` = 15 bytes) to the right and be masked. The comment inside the assembly block has wrongly stated 12 bytes of the `loc` instead of `len`.

Code partially corrected:

The reported inconsistencies 1-4 have been fixed in the updated codebase, while the last one remains unchanged.

5.4 Missing Sanity Checks

Design | Low | Version 1 | Code Partially Corrected

The following functions set important state variables or parameters, but do not perform any sanity check on input parameters:

1. `MessageTransmitter.constructor`.
2. `TokenMessenger.constructor`.
3. `MessageTransmitter.setMaxMessageBodySize`.
4. `newMintRecipient` in `TokenMessenger.replaceDepositForBurn`.

Code partially corrected:

Sanity checks were added in `TokenMessenger.constructor` and `TokenMessenger.replaceDepositForBurn` listed above, however no sanity checks were added for points 1 and 3.

5.5 Potential Event Reordering Due to Reentrancy in MessageTransmitter

Security | Low | Version 1 | Acknowledged

The function `sendMessage` does not have any access restriction, and the caller can pass any arbitrary value for `recipient`. On the other side of the bridge, the function `receiveMessage` gives execution to `recipient` and emits an event afterward. Therefore, a malicious `recipient` could reenter the contract causing events to be emitted in an inconsistent order:

```
require(
    IMessageHandler(_m._recipientAddress()).handleReceiveMessage(
        _sourceDomain,
        _sender,
        _messageBody
    ),
    "handleReceiveMessage() failed"
);

// Emit MessageReceived event
emit MessageReceived(
    msg.sender,
    _sourceDomain,
    _nonce,
    _sender,
    _messageBody
);
```

Acknowledged:

Circle acknowledged the issue but has decided to keep the code unchanged.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	6

- Default Optimizer Configuration Code Corrected
- Inconsistent Type Used for Nonce Code Corrected
- Missing Event in Ownable Code Corrected
- Unchecked Return Value for Functions From TypedMemView Code Corrected
- Unrelevant Indexed Event Fields Code Corrected
- Wrong Values Emitted in Event Code Corrected

6.1 Default Optimizer Configuration

Design Low Version 1 Code Corrected

The compiler optimizer is not enabled explicitly by the foundry configuration, hence the default optimizer enabled by the foundry with 200 runs is used:

```
[profile.default]
src = 'src'
out = 'out'
libs = ['lib']
```

The optimizer uses the specified number of runs to perform a trade-off between deployment cost (bytecode size) versus execution costs. A high number of runs indicates to the optimizer that the reduction of execution costs has a higher priority than deployment costs.

Code corrected:

The configuration file `foundry.toml` has been updated to enable the optimizer with `10_000` runs.

6.2 Inconsistent Type Used for Nonce

Design Low Version 1 Code Corrected

The contract `MessageTransmitter` uses type `uint64` for storing nonces, however, the internal function `_hashSourceAndNonce` uses `uint256` for the argument `_nonce`.

Code corrected:

Type of _nonce in _hashSourceAndNonce is changed to uint64 and is consistent throughout the code.

6.3 Missing Event in Ownable

Design Low Version 1 Code Corrected

The constructor of Ownable sets the deployer of the contract as owner, however, the respective event is not emitted.

Code corrected:

The constructor of Ownable now calls the internal function _transferOwnership which sets the new _owner and emits the respective event.

6.4 Unchecked Return Value for Functions From TypedMemView

Design Low Version 1 Code Corrected

The functions ref and slice of the library TypedMemView return a memory view of type bytes29. However, both functions can return NULL which represents an invalid type (ff_ffff_ffff) if the memory is malformed. The calling functions in MessageTransmitter, TokenMessenger and Message do not check for the invalid type.

Code corrected:

The libraries Message and BurnMessage have been extended with functions _validateMessageFormat and _validateBurnMessageFormat. These functions are now used to validate the return values from functions ref and slice from the library TypedMemView.

6.5 Unrelevant Indexed Event Fields

Design Low Version 1 Code Corrected

Only relevant fields of the events should be indexed, the ones which it makes sense to search for. The following events index also uint values:

1. amount in TokenMessenger.DepositForBurn
2. amount in TokenMessenger.MintAndWithdraw
3. oldSignatureThreshold and newSignatureThreshold in Attestable.SignatureThresholdUpdated
4. burnLimitPerTransaction in TokenController.SetBurnLimitPerTransaction
5. newMaxMessageBodySize in MessageTransmitter.MaxMessageBodySizeUpdated

On the other hand, the event `OwnershipTransferred` does not index its argument. EVM opcodes for logging events with more indexed arguments consume more gas. We suggest for each event field reevaluate if indexing is necessary.

Code corrected:

All events listed above were revised such that `uint` arguments are no longer indexed.

6.6 Wrong Values Emitted in Event

Correctness **Low** **Version 1** **Code Corrected**

Function `updateAttesterManager` uses the same variable `newAttesterManager` in the emitted event. The natspec of the event specifies that the first parameter is the address of the previous attester manager, while the second parameter is the new attester manager.

Code partially corrected:

The function `updateAttesterManager` has been revised in Version 2 to pass `msg.sender` and `newAttesterManager` as parameters to the event `AttesterManagerUpdated`. However, the first parameter `msg.sender` is the owner of the contract, and not necessarily the previous manager as described in the event definition.

Code corrected:

In Version 3, the following code is used to emit the previous and new addresses for the `attesterManager` role:

```
address _oldAttesterManager = _attesterManager;
_setAttesterManager(newAttesterManager);
emit AttesterManagerUpdated(_oldAttesterManager, newAttesterManager);
```

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Compiler Version Not Fixed and Outdated

Note Version 1

The solidity compiler is fixed only in contracts `Ownable`, `Pausable` and `Rescuable`, while other contracts use the following pragma directive:

```
pragma solidity ^0.7.6;
```

Although no later compiler version `0.7.x` exist, it is a best practice to fix the compiler version in contracts or configuration file.

Known bugs in version `0.7.6` are listed here.

More information about these bugs can be found here: https://docs.soliditylang.org/en/latest/bugs.html

At the time of writing the most recent Solidity release is version `0.8.17` which contains some bugfixes. However, version `0.8` introduced breaking changes and would require heavy refactoring of the contracts.

Version 2 changes: All contracts now use the following pragma directive:

```
pragma solidity 0.7.6;
```

7.2 Non-canonical Conversion of Bytes to Address

Note Version 1

The function `Message.bytes32ToAddress` implements the following statement to perform the type conversion:

```
function bytes32ToAddress(bytes32 _buf) public pure returns (address) {
    return address(uint160(uint256(_buf)));
}
```

Note that due to downcasting, higher bits of `_buf` will be omitted. Thus, it is possible to have different input values `_buf` map to the same `address`.

Version 2 changes: Circle has decided to emphasize this behavior in the code by appending the following description to the function's natspec:

```
* @dev Warning: it is possible to have different input values _buf map to the same address.
* For use cases where this is not acceptable, validate that the first 12 bytes of _buf are zero-padding.
```

7.3 Overflow and Underflow Occurring in TypedMemView

Note Version 1

The function TypedMemView.index takes as the third argument the length of the returned value in bytes _bytes, which is of type uint8. The length in bits is computed as follows:

```
uint8 bitLength = _bytes * 8;
```

If _bytes is 32, the multiplication above overflows as the result 256 cannot be stored in a variable of type uint8, hence bitLength stores 0. Furthermore, when bitLength is passed to function leftMask an underflow occurs in the following assembly code:

```
assembly {
    mask := sar(
        sub(_len, 1),
        ...
    )
}
```

7.4 Potential Single Points of Failure

Note Version 1

Circle EVM Bridge relies on a centralized attestation service (attesters) to guarantee the integrity of messages transmitted between chains. The protocol assumes that an adversary cannot compromise enough attesters (signatureThreshold) at the same time, otherwise, the bridge becomes vulnerable.

Besides the assumption above, we would like to highlight below the accounts that are potential single points of failure for the security of the bridge.

Message Transmitter: Any account with role owner or attesterManager should be carefully protected. If any account with these roles gets compromised, it can freely enable new attesters and execute arbitrary cross-chain messages. Furthermore, the role pauser is critical to be protected in order to keep the bridge operational and avoid denial-of-service (DoS) attacks.

Token Messenger: The account with the role owner should be carefully protected, as if this account gets compromised, it can set arbitrary addresses as token messengers in remote domains and then process malicious messages.

Token Minter: The accounts with roles owner and tokenController should be carefully protected. If any of these accounts get compromised, the mapping remoteTokensToLocalTokens can be manipulated, which can consequently create severe issues, e.g., an attacker can burn low value tokens in one chain but mint the same amount in high value tokens in the other chain.

7.5 Return Value of Burn Function

Note Version 1

The system supports tokens that implement the IMintBurnToken, i.e., functions transfer, transferFrom and mint return a boolean value. However, burn function is assumed to not return a value but revert if unsuccessful. This behavior is in line with the implementations of `USDC` and `ERC20Burnable` from OpenZeppelin.

7.6 Signature Threshold Restrictions

Note **Version 1**

The documentation states that the threshold for the required signatures should not be below 2, however, this is not enforced by the codebase. On deployment, the constructor of `Attestable` contract takes only one `attester` address as an argument and sets `signatureThreshold = 1`.

Furthermore, the function `setSignatureThreshold` does not enforce that the threshold is set to at least 2. Circle is aware of this behavior and does not intend to enforce the minimum threshold in code.

7.7 Visibility Modifiers for Constructors

Note **Version 1**

Contracts `Attestable` and `Ownable` declare the visibility of constructors as `public`, however, such visibilities in compiler version `0.7.6` are obsolete. More information.

Version 2 changes: The visibility for constructors has been removed in the updated codebase.

7.8 `Attestable._recoverAttesterSignature` Function Visibility Can Be Pure

Note **Version 1**

The modifier of the function `_recoverAttesterSignature` can be changed to `pure`, as it neither writes nor reads the storage of the contract.

Version 2 changes: The visibility of the function above has been changed to `pure`.