PUBLIC

Code Assessment

of the CCTP V2 Smart Contracts

March 24, 2025

Produced for

by

Contents

1 Executive Summary

Dear Circle Team,

Thank you for trusting us to help Circle with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of CCTP V2 according to Scope to support you in forming an opinion on their security risks.

Circle implements a new version of Cross-Chain Transfer Protocol (CCTP) that implements new features, such as fast burn messages and hook data. CCTP v2 uses new message formats that are incompatible with the previous version. Two versions of CCTP are treated as distinct networks in the smart contract level and use a different set of attesters.

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

The general subjects covered are functional correctness, upgradability, trustworthiness, and documentation. The core contracts in the scope of this review are upgradeable and fully controlled by an admin role. In addition, several accounts are required to be trusted, see Roles and Trust Model and 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	1
• Code Corrected	1
Low-Severity Findings	7
• Code Corrected	4
• 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 v2 related source code files inside the CCTP V2's src repository based on the documentation files. Additionally, the relevant functions used from the `TypedMemView` library contract were assessed. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	25 Nov 2024	5ee9bbd27dbdc479c3d5dd2b024b87ecb1b167f3	Initial Version
2	20 Jan 2025	7d703109a2cfcb3f76375fef5f1a97f03c447b94	Final Version

For the solidity smart contracts, the compiler version `0.7.6` was chosen. The following files were in the scope of this review:

```
interfaces/v2/*
roles/v2/Denylistable.sol
roles/v2/AttestableV2.sol
roles/Attestable.sol (only diff with previously audited commit)
roles/Ownable.sol (only diff with previously audited commit)
roles/Ownable2Step.sol (only diff with previously audited commit)
roles/Pausable.sol (only diff with previously audited commit)
roles/Rescuable.sol (only diff with previously audited commit)
roles/TokenController.sol (only diff with previously audited commit)
proxy/AdminUpgradableProxy.sol
proxy/Initializable.sol
v2/Create2Factory.sol
v2/BaseTokenMessenger.sol
v2/BaseMessageTransmitter.sol
v2/MessageTransmitterV2.sol
v2/TokenMessengerV2.sol
v2/TokenMinterV2.sol
v2/FinalityThresholds.sol
TokenMinter.sol (only diff with previously audited commit)
examples/CCTPHookWrapper.sol
```

2.1.1 Excluded from scope

All used third-party code like libraries are excluded from the scope. Except for the relevant functionality that is used in the code base from the `TypedMemView` library. The system heavily relies on off-chain operations. All operations not performed and done by smart contracts that might introduce or lead to vulnerabilities are excluded from the scope. As a bridge, the same code will be deployed on many different blockchains. These chains differ in their behavior and their behavior might change in the future. The assessment was done on Ethereum's behavior and EVM specifications. Before deploying on another chains, the compatibility needs to be assessed and tested thoroughly. The same applies for supported tokens to be bridged as some might be incompatible with the current setup (e.g., re-basing tokens, tokens with fees on transfers, etc.).

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. The core functionality is to send messages from one chain to another. On top of this, the bridge also has a pre-defined flow to transfer tokens across chains. The core and generic message transmission can also be used to design different token bridges or applications.

Three steps are required for a generic message to be transferred across the chain:

1. user calls `sendMessage()` in `MessageTransmitterV2`, 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.

The CCTP protocol design can be abstracted into two layers:

- Transmission layer: `MessageTransmitterV2` exposes functionalities to users to send arbitrary messages to any recipient address in a destination domain (chain).
- Logic layer: `TokenMessengerV2` sends and receives messages from `MessageTransmitterV2` 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.

This review did an assessment of the newly added v2 features. The overall system architecture has remained the same. The most important changes are:

- The addition of new fields in generic messages and burn messages.
- The use of upgradeable proxy contracts for the transmitter and messenger contracts.
- A new feature allowing hooks to be called with a message transmission.
- A denylist that can disallow certain addresses from initiating cross-chain messages via `TokenMessengerV2`. However, such addresses can still consume messages in the destination domain.
- A new functionality allowing faster message transmission for a premium (fee).

2.2.1 Message Transmitter

The message transmitter contract v1 was updated and split into two contracts in v2. The two contracts are the `BaseMessageTransmitter` and the `MessageTransmitterV2`. The base contract is used for administration and configuration functions and currently has the `setMaxMessageBodySize` function to set the maximum message body size. Besides, the contract inherits the relevant administrative contracts `Initializable` (for upgradability), `Pausable` (allowing to pause the message transmission), `Rescuable` (to recover funds send to the contract) and `AttestableV2` (to manage the set of attesters and attestation configuration).

The `MessageTransmitterV2` contract implements the core functionality of the protocol's transmission layer 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 and a version to have a versioning system for the implementation contracts that are called by the new upgradable proxy setup.

`MessageTransmitterV2` 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 field for the nonce that is checked in the destination domain. Attesters set the nonce, ensuring it is unique to each message. By specifying a finality threshold parameter, the user can choose if they want to send a fast message (unfinalized and with fees) or the normal finalized message. The confirmation time determines the message speed. Additionally, a `destinationCaller` can be defined who is allowed to call `receiveMessage()` on destination domain. Whereas the `recipient` is the address that is called by the `MessageTransmitterV2` (either `handleReceiveUnfinalizedMessage()` or `handleReceiveFinalizedMessage()` is called on the `recipient`). If the message is formatted correctly, the event `MessageSent` is emitted.
- `receiveMessage`: anyone can call this function to broadcast a cross-chain message 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 `handleReceiveUnfinalizedMessage` on the `recipient` address if the `_finalityThresholdExecuted` is below `2000` (implying it shall be a fast, not completely finalized message). If the threshold is set to `2000` (or higher), the message will be treated as a normally finalized message and `handleReceiveFinalizedMessage` is called.

The attestation service is composed of a set of centralized servers which listen to `MessageSent` events emitted by `MessageTransmitterV2` and sign them upon user request.

Importantly, the set of attestors providing signatures for messages of version 2 should not overlap with attesters providing signatures for messages version 1, see CCTP v2 should use different attesters from CCTP v1. Further, the two versions of CCTP use different message formats, hence they should be treated as distinct and independent networks.

2.2.2 Token Messenger

The token messenger contract operates on the logic layer of the protocol. The updated v2 is split into two contracts, too. The `BaseTokenMessenger` and the `TokenMessengerV2`. The base contract is used for administration and configuration functions and currently has the messenger management, minter management and fee recipient management functions:

- `addRemoteTokenMessenger` and `removeRemoteTokenMessenger`: will associate or disassociate a remote token messenger contract in the mapping `remoteTokenMessengers` with the domain where the remote token messenger is deployed as key.
- `addLocalMinter` and `removeLocalMinter`: will associate or disassociate a local minter contract in the `localMinter` variable. The contract is called to mint and burn tokens.
- `setFeeRecipient` sets the fee recipient for quick messages.

Additionally, the base contract has the internal `_depositAndBurn` and `_mintAndWithdraw` functions that are called by the `TokenMessengerV2` contract to mint and burn the tokens. Besides, the contract inherits the relevant administrative contracts `Initializable` (for upgradability), `Rescuable` (to recover funds send to the contract) and `Denylistable` (allowing to manage a list of accounts that are blocked from using the token messenger contract).

The `TokenMessengerV2` has the user facing functions to interact with the bridge:

- `depositForBurn` and `depositForBurnWithHook`: everyone can call these functions to burn funds and call the transmitter contract to emit an event that the attestation service will use to sign the message such that it can be used in the receiver functions on the other chain. The caller specifies the number of tokens to be bridged, the destination domain, a mint receiver, the token contract, the recipient address in the destination domain, a destination caller who is allowed to call the receive function on the other chain. The new functionality also allows to specify if the message shall be sent via the new "fast" feature. In this case a maximum fee can be defined by the user. The function burns tokens in the source domain (chain) and composes a `BurnMessage`, which is forwarded to `sendMessage` in `MessageTransmitterV2`.
- `depositForBurnWithHook`: allows to send additional data to be called by the hook contract on the receiver chain.
- `handleReceiveFinalizedMessage` and `handleReceiveUnfinalizedMessage` can be called only by a message transmitter contract 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; hence, `handleReceive**Message` only checks that the received `messageBody` is a valid `BurnMessage`. If successful, a call to the `localMinter` is made to mint the respective tokens. For an unfinalized message, the function will check if the message is not expired, and the fee does not exceed the user defined maximum fee.

Unfinalized messages, referred as fast messages, have a higher risk for the bridge due to possible reorgs in the source chain. If a fast burn message is signed by attesters, and then a reorg happens on the chain, the bridge suffers losses. Attesters are not obliged to always sign fast messages as they are subject to other off-chain limitations. For instance, fast burn messages are covered by an off-chain insurance fund and available only to whitelisted entities.

On deployment, it is linked to the correct instance of `MessageTransmitterV2`. The `BaseTokenMessenger` does not inherit `Pausable`. Hence, the contract cannot be paused.

2.2.3 Token Minter

The approach used by CCTP is to burn tokens in one chain and mint them in the other chain. This approach is different from the bridges that 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. The updated version `TokenMinterV2` inherits all functionality from the version 1 contract and implements a new minting function such that it allows to specify two mint operations with two different recipients instead of just one as in the first version. The contract is not upgradable like other v2 contracts.

The `TokenMinter` implements the following functionalities:

- `mint`: this function can be called only by the local `TokenMessengerV2` and mints the respective tokens to the two specified recipient address `recipientOne` and `recipientTwo`. 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 `TokenMessengerV2` and implements another rate limiting mechanism, which allows burning tokens up to a threshold set by the `tokenController`.

Furthermore, the `TokenMinter` inherits the relevant administrative contracts `Pausable` (allowing to pause the message transmission), and `Rescuable` (to recover funds send to the contract) and `TokenController` (managing the burn amounts and linking the tokens to the remote domains).

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.

In v2 burning in the local domain requires the following steps:

1. The user calls `depositForBurn` or `depositForBurnWithHook` on the local `TokenMessengerV2`.
2. `TokenMessengerV2` calls into the local `TokenMinterV2` to burn the user's local token.
3. `TokenMessengerV2` creates a `BurnMessageV2`, including the maximum fee for fast messages and hook data if present.
4. After formatting the `BurnMessageV2`, `TokenMessengerV2` calls `sendMessage` in `MessageTransmitterV2`.
5. `MessageTransmitterV2` eventually encapsulates this `BurnMessageV2` as a payload body 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. The user can specify in version 2 how fast the message shall be attested. A faster attestation has less confirmations on the chain and costs a fee. In case the fast attestation was desired but not possible, the attesters will treat it as a normal message and not charge the fee. After fetching the attestation from a minimum pre-defined number of whitelisted attesters, the signature alongside the original message should be fed into the `MessageTransmitterV2` in the destination domain:

1. Caller triggers the minting operation in the destination domain by feeding in the signed message to function `receiveMessage` in `MessageTransmitterV2`.
2. After verifying the signatures, `MessageTransmitterV2` calls `handleReceiveUnfinalizedMessage` or `handleReceiveFinalizedMessage` of the recipient (the respective `TokenMessengerV2`).
3. Eventually, `mint` in the token contract is called by `TokenMinterV2` in the destination domain, which mints tokens with a 1:1 ratio with burned tokens.

2.4 Create2Factory

This factory will be deployed at the same address on different chains by Circle and afterwards will be used to deploy the proxies for the message transmitter and the token messenger via `create2`. Therefore, the proxies are deployed at a deterministic address in different chains. Only `owner` can deploy contracts via this factory.

For the deployment of proxies, the function `deployAndMultiCall()` should be used in order to initialize the proxies with correct roles immediately after deployment. Otherwise, an attacker can take over the proxies and assign privileged roles to untrusted accounts.

2.5 Roles and Trust Model

All contracts presented above have privileged accounts that need to be trusted to behave correctly for the protocol to function as expected. In general, we assume the deployers and accounts with the `owner` or `admin` roles are fully trusted and they configure contracts with the correct parameters. Most importantly, admins of proxies are considered to be fully trusted and behave in the best interests of the system. We detail these accounts below.

Message Transmitter

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

- `deployer` and `initializer`: sets the immutable variables `localDomain` and `version` of the contract. Additionally, the initializer sets critical variables like:
    - `owner`,
    - `pauser`,
    - `rescuer`,
    - `attesterManager`,
    - `attesters`,
    - `signatureThreshold`,
    - `maxMessageBodySize`
- `owner`: can transfer ownership to another account, update the maximum size for the message body, and set addresses for `rescuer`, `newAttesterManager` 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.

Token Messenger

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

- `deployer` and `initializer`: sets the immutable variables `localMessageTransmitter` and `messageBodyVersion`. Additionally, the initializer sets critical variables like:
    - `owner`
    - `rescuer`
    - `feeRecipient`
    - `denylister`
    - `localMinter`
    - `remoteTokenMessengers`
- `owner`: can change the ownership, sets the respective addresses of remote token messenger contracts, can update the address serving as `localMinter`, the denylist admin 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.
- `Denylister`: manages the denylist of addresses that are not allowed to interact with the bridge. This account can add or remove addresses from the denylist and transfer the role to another account.

- `rescuer`: 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.

Token Minter

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

- `deployer`: sets the `tokenController` address that can be re-set by the owner.
- `owner`: sets and removes address of the local `TokenMessengerV2` 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.
- `rescuer`: 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.

Attestation Service

The whitelisted servers (attesters), that compose the attestation service, are trusted to sign only authentic messages emitted by the `MessageTransmitterV2` contract. In this review, we assume attesters sign messages only for events that have been emitted in finalized blocks. Attesting to messages from unfinalized blocks come with additional risks related to potential reorgs, therefore we assume such messages are covered by the off-chain insurance fund.

Upgradeability

The contracts in the scope of this review are partly upgradeable; We assume there is only one version of contracts `MessageTransmitterV2` and `TokenMessengerV2` 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 second version of the contract, any future version is not in the scope of this review.

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.

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, fees on transfer, etc.) and implementing the `IMintBurnToken` interface are supported by the system. Finally, supported tokens should also implement the function `burn` which reverts on failure.

Users

Users interacting with the system are assumed to be untrusted.

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 Open 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	3

5.1 Potential Event Reordering Due to Reentrancy

Security Low Version 1 Acknowledged

CS-EVM-CCTP2-001

The function `sendMessage()` does not have any access restriction, and the caller can pass any arbitrary value for `recipient`. On the destination chain, the function `receiveMessage` passes 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
);
```

Similarly, the token minter passes the execution to an external token when calling `token.mint()`. If `token` implements hooks on minting, then `mintRecipient` can potentially reenter in the contracts.

Acknowledged:

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

5.2 Setting Max Burn Amount Without Linked Token

Design | Low | Version 1 | Acknowledged

CS-EVM-CCTP2-002

It is possible to set the max burn amount in `TokenController.setMaxBurnAmountPerMessage` even without a linked token. If a burn limit is set for an unlinked token, users can accidentally burn their tokens in the source domain although they cannot be minted in the destination domain.

Acknowledged:

Circle replied:

```
This is ultimately a misconfiguration, and there are a variety of existing off-chain
processes for guarding against this. These processes were developed for CCTP v1,
and preserving the same functions in v2 allows us to continue leveraging these
protections.
```

5.3 Type 0 Used for All Memory Views

Design | Low | Version 1 | Acknowledged

CS-EVM-CCTP2-003

All memory views created through the library `TypedMemView` use `type 0`. For instance, contract `MessageTransmitterV2` uses `type 0` for the memory view referencing a `MessageV2` in function `_validateMessageFormat()`. The same memory view `type 0` is used in function `MessageV2._getMessageBody()` although the view now references a message body.

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 Open 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	1

- Fees for Fast Burns Can Be Avoided Code Corrected

Low-Severity Findings	4

- Incorrect Natspec Code Corrected
- Missing Events on Initialization Code Corrected
- Old Version of TypedMemView Code Corrected
- Possible Underflows in Relay Function Code Corrected

Informational Findings	2

- Redundant Casting Code Corrected
- Use Constant Instead of 1 or 0 Code Corrected

6.1 Fees for Fast Burns Can Be Avoided

Correctness Medium Version 1 Code Corrected

CS-EVM-CCTP2-016

Function `receiveMessage()` in contract `MessageTransmitterV2` checks the status of attestations for a message. The message field `finalityThresholdExecuted` indicates waiting time before the attestation for a message is provided. Fast burn messages, attested more quickly than normal messages, should pay a fee that is collected by `TokenMessengerV2` when the message is consumed in the destination chain. However, in case a user requests attestations for a fast burn message but does not consume the message before expiry, no fee is charged. The user can still request for re-attestation after expiry, but the field `finalityThresholdExecuted` would indicate a normal message.

Code corrected:

The function `handleReceiveFinalizedMessage()` in contract `TokenMessengerV2` has been revised to handle regular messages that should pay a fee.

Depending on user's specified finality, attesters fill the fields `expirationBlock` and `feeExecuted` as follows:

Regular messages:

```
expirationBlock = 0
feeExecuted = 0
```

Fast-burn messages:

```
expirationBlock = x; x != 0
feeExecuted = x; x >= 0
```

If a message is initially attested as a fast-burn, but expires before submitted on the destination chain, can be re-attested as a regular message with fields set as follows:

```
expirationBlock = 0
feeExecuted = x (same value it was originally set to when it was signed as a fast message)
```

6.2 Incorrect Natspec

Correctness Low Version 1 Code Corrected

CS-EVM-CCTP2-015

The natspec description for function `linkTokenPair()` lists a requirement which refers to a function which is not implemented:

```
Setting a token pair does not enable the `localToken` (that requires calling setLocalTokenEnabledStatus.)
```

Code corrected:

The natspec comment has been corrected to refer to function `setMaxBurnAmountPerMessage()`.

6.3 Missing Events on Initialization

Design Low Version 1 Code Corrected

CS-EVM-CCTP2-014

Functions `initialize()` in MessageTransmitterV2 and TokenMessengerV2 do not emit all events when important state values are set. For instance, `MessageTransmitterV2.initialize()` does not emit an event when setting `_attesterManager`. Similarly, `TokenMessengerV2.initialize()` does not emit the respective events when configuring the remote messengers.

Code corrected:

The functions `initialize()` have been revised to emit the relevant events.

6.4 Old Version of TypedMemView

Design Low Version 1 Code Corrected

CS-EVM-CCTP2-011

The codebase uses an old version of the dependency `TypedMemView`. More recent versions of the library are available which include bug fixes.

Code corrected:

The codebase Version 2 uses the latest version of the dependency `TypedMemView`.

6.5 Possible Underflows in Relay Function

Design | Low | Version 1 | Code Corrected

CS-EVM-CCTP2-013

`CCTPHookWrapper` is supposed to consume messages of type `BurnMessageV2`, however one can call the function `relay()` for a message with a different format. In such cases, the call to `_getHookData()` causes an underflow when computing the end index of the slicing range. Similarly, the call to `_getMessageBody()` might underflow.

Code corrected:

The function `relay()` has been revised to perform checks for lengths and versions of both `MessageV2` and `BurnMessageV2` before calling `_getHookData()`.

6.6 Redundant Casting

Informational | Version 1 | Code Corrected

CS-EVM-CCTP2-010

Function `_isLocalMessageTransmitter()` in BaseTokenMessenger performs an unnecesary casting in `address(localMessageTransmitter)` as the variable is already an `address`.

Code corrected:

The unnecesary casting has been removed in Version 2.

6.7 Use Constant Instead of 1 or 0

Informational | Version 1 | Code Corrected

CS-EVM-CCTP2-012

To invalidate a nonce the value 1 is assigned to the `usedNonces` mapping. To improve readability and clarity a constant called e.g., `NONCE_USED`, could be used. This is usually done e.g., in the reentrancy protection with `_NOT_ENTERED = 1`.

Code corrected:

The constant `NONCE_USED = 1` has been added in Version 2.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Overflow and Underflow Occurring in TypedMemView

Informational **Version 1**

CS-EVM-CCTP2-004

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),
        ...
    )
}
```

7.2 Repeated Events Possible

Informational **Version 1**

CS-EVM-CCTP2-005

Function `Denylistable.denylist()` does not check if `account` is already in the deny list, therefore the event `Denylisted` can be emitted repeatedly for the same address.

7.3 Roles Set in Implementation Contract

Informational **Version 1**

CS-EVM-CCTP2-006

MessageTransmitterV2 is deployed behind a proxy, and upon deployment of the implementation contract the immutable variables are set. Furthermore, the constructor of MessageTransmitterV2 triggers a call to `Attestable.constructor()` which sets the attester manager, threshold, and the first attester in the implementation contract. However, these values live in the implementation contract and are not used by the proxy.

7.4 View Functions in Proxy

Informational Version 1

CS-EVM-CCTP2-007

View functions `admin()` and `implementation()` do not use the modifier `ifAdmin`, therefore they return the respective addresses stored in the storage of the proxy at slots `_ADMIN_SLOT` and `_IMPLEMENTATION_SLOT` for any caller.

In case implementation contracts implement functions with same selectors as `admin()` or `implementation()`, they are not reachable externally.

7.5 Known Issues in OpenZeppelin Dependency

Informational Version 1 Acknowledged

CS-EVM-CCTP2-008

The codebase uses Solidity compiler version `0.7.6` and therefore uses `v3.4.2` of OpenZeppelin contracts as its dependency. This version has known issues that are listed in the link: https://github.com/OpenZeppelin/openzeppelin-contracts/security.

Acknowledged:

Circle replied:

```
We do not believe that any of the known OpenZeppelin issues are relevant
to CCTP v2 code as it is currently written.
```

7.6 No Event Index

Informational Version 1 Acknowledged

CS-EVM-CCTP2-009

The events `TokenPairLinked` and `TokenPairUnlinked` work as specified. To ensure no indexing is needed, we emphasize in this issue that both have no indexing of their fields to allow easy filtering for the `localToken`.

The same applies to:

- `AdminUpgradableProxy.AdminChanged`
- In `BaseTokenMessenger` `LocalMinterAdded`, `LocalMinterRemoved` and `FeeRecipientSet`

Acknowledged:

Circle replied:

```
These events are emitted extremely infrequently from a fixed set of contracts,
so adding indexed fields for localToken does not add much value.
```

8 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.

8.1 Attesters Should Fill Empty Fields With Same Values

Note Version 1

CCTPv2 uses message formats that include fields that are expected to be empty when the event `MessageSent` is emitted. The empty fields are:

- Fields `nonce` and `finalityThresholdExecuted` in MessageV2.
- Fields `feeExecuted` and `expirationBlock` in BurnMessageV2.

Attesters are responsible for filling the empty fields before providing the attestations for a message. Attesters should fill the messages with same values, hence sign the same hash, for the messages to be consumed in the destination chain.

8.2 CCTP V2 Should Use Different Attesters From CCTP V1

Note Version 1

CCTP v2 maintains the same architecture with CCTP v1 and possibly the same attestation service can be used for both versions. We emphasize that this setup enables security vulnerabilities and is forbidden. The core vulnerability relies on the fact that CCTP v1 implements a message replacement functionality that allows users to update specific fields of a message if valid signatures are provided.

Functions `MessageTransmitter.replaceMessage()` and `TokenMessenger.replaceDepositForBurn()` (from CCTP v1) always replace the message version. Therefore, if the same attesters were to be used for both versions of CCTP, one could exploit this functionality of CCTP v1 to modify messages intended for CCTP v2, hence generating messages that violate the specifications.

8.3 Chain Reorgs Risk Fast Messages

Note Version 1

Fast messages have a higher risk for the bridge as the chain might reorg after the attestation is produced. This is an acceptable risk that is covered by the off-chain insurance fund. An entity that controls multiple blocks in a row and is eligible for fast messages, can exploit this feature, to initiate a fast cross-chain message and trigger a reorg afterwards.

8.4 Expiration of Fast Messages Relies on Block.Number

Note Version 1

Function `TokenMessengerV2.handleReceiveUnfinalizedMessage()` uses `block.number` to check if a fast burn message has already expired.

We highlight that `block.number` might behave differently on different chains. For instance, Arbitrum, `block.number` returns the estimated L1 block number.

Circle replied:

```
We will remain using block.number, and ensure that any differing behavior with
Arbitrum's block.number calculation is handled through the off-chain service that
provides the expirationBlock value
```

8.5 Finality Thresholds

Note Version 1

In the file FinalityThresholds.sol, there are three thresholds declared for message finality:

```
// The threshold at which (and above) messages are considered finalized.
uint32 constant FINALITY_THRESHOLD_FINALIZED = 2000;

// The threshold at which (and above) messages are considered confirmed.
uint32 constant FINALITY_THRESHOLD_CONFIRMED = 1000;

// The minimum allowed level of finality accepted by TokenMessenger
uint32 constant TOKEN_MESSENGER_MIN_FINALITY_THRESHOLD = 500;
```

The codebase intentionally does not perform sanity checks to allow users choose any finality threshold. However, for this report we assume any threshold in the range [500, 1000] is considered as fast burn message by attesters and `finalityThresholdExecuted` is set to 1000. If the chosen threshold is above 1000 (can be above 2000 also), the message is considered as normal message by attesters and its `finalityThresholdExecuted` is set to 2000.

8.6 Pattern for Changing Admin Role of Proxies

Note Version 1

The codebase follows the 2-step pattern when transferring roles of high importance to avoid lockouts. However, this pattern is not followed for the `admin` role of proxies, hence extra measures should be taken to avoid lockouts.

8.7 Potential Single Points of Failure

Note Version 1

CCTP V2 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.

Proxies: Accounts with role `admin` for `MessageTransmitterV2` and `TokenMessengerV2` proxies are considered fully trusted. If any of these accounts is compromised, they can upgrade a proxy to a malicious implementation.

`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. Furthermore, the role `denylister` should be protected to keep the bridge accessible to integrators and avoid DoS attacks.

`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.