certora

Security Assessment

Mayan

March 2025

Prepared for Mayan

Table of content

Project Summary..3

Project Scope....3
Project Overview....3
Protocol Overview....3
Findings Summary....4
Severity Matrix....4
Detailed Findings....5
Medium Severity Issues....6
M-01 A malicious guardian can DoS the bridge in several ways....6
Low Severity Issues....8
L-01 MIN_FINALITY_THRESHOLD is not set correctly....8
L-02 Guardian can replace the fee manager to steal the redeem fee....9
Informational Severity Issues....10
I-01. Incorrect revert reason handling in low-level call....10
I-02. Deprecated safe approval method used in approval pattern....11
I-03. Origin of cross-chain messages is not validated....13
I-04. Potential gas manipulation in eth transfers due to 1/64th rule....14
I-05. Add events for guardian-controlled functions....16
I-06. depositRelayerFee should not be called when redeemFee = 0....17
I-07. Implement Time-Delay for Guardian Functions....18
I-08. Typos in variable naming....19
I-09. Incomplete Events: OrderFulfilled & OrderRefunded....20
I-10. Unused Errors....21
I-12 User can use the refund() function to avoid paying protocol fees....23
I-13 User can use redeem fee to avoid paying protocol fees....24
I-14 Hardcoded eth decimals limit cross-chain compatibility....25
I-15 Common user errors might lead to funds stuck on chain....27
I-16 Refund event amount includes refund fee....28
Disclaimer....29
About Certora....29

Project Summary

Project Scope

Project Name	Repository (link)	Audit Commit Hash	Fix commit hash	Platform
Mayan	Mayan	e46d35751aa51316f426f9e4b2dbc32fc6204ee2	416affdf77239bd084a96535bdd3fb27ae52849d	EVM

Project Overview

This document describes the specification and verification of **Mayan** manual code review findings. The work was undertaken from **March 3rd** to **March 10th**

The following contract list is included in our scope:

```
 -  src/FastMCTP.sol
```

Protocol Overview

Mayan is a cross-chain swap auction protocol with the goal of offering the best swap rates using auction mechanisms.

It also offers Mayan-Circle Transfer Protocol (MCTP) using CCTP as the highway for transferring the value to the destination chain, and on the destination chain the driver fulfills the user's trade using Mayan Flash Swap smart contract/program and delivers the output tokens.

The upcoming CCTPv2 functionality implementation for Mayan is the subject of this audit.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	1	1	1
Low	2	1	1
Informational	15	-	-
Total	18	2	2

Severity Matrix

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

Detailed Findings

ID	Title	Severity	Status
M-01	A malicious guardian can DoS the bridge in several ways	Medium	Fixed
L-01	MIN_FINALITY_THRESHOLD is not set correctly	Low	Fixed
L-02	Guardian can replace the fee manager to steal the redeem fee	Low	Acknowledged

Medium Severity Issues

M-01 A malicious guardian can DoS the bridge in several ways		
Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: src/FastMCTP.sol	Status: Fixed	

Description: The guardian can set the fee manager contract via the `setFeeManager()` function. This replaces the fee manager instantly without any delay.
A malicious guardian can do any of the following:

- Set the fee manager to a contract that would revert all calls to it
  - This would DoS both `redeem()` and `fulfillOrder()`
- Set the fee manager to a blacklisted address
  - this would revert the deposit of the redeem fee to the fee manager and subsequently revert the `redeem()` function
- Set the protocol fee to a high value
  - This would affect `fulfillOrder()` - either charging high fees for orders, or reverting the call if it leaves the user with less than min amount out

Exploit Scenario:

- Guardian wallet gets compromised by an attacker
- Attacker sets themselves as the guardian
- Attacker replaces the fee manager with the zero address, all calls to it would be reverted
- All funds bridged to FastMCTP are now stuck, causing a loss of funds for users

Recommendations:

- Any calls to the fee manager should be done as an excessively safe call

- Transfer to the fee manager should be inside a try-catch block
- Limit the protocol fee to 100 bps for `fulfillOrder()`

Customer's response: Fixed in PR #14 (5730bad..79f77c)

Fix Review: Fix confirmed. During the fix review we've ensured that calls to the fee manager can't revert by either using an excessive amount of gas or by returning a value that would cause `abi.decode()` to revert. The current fix addresses all of that.

Low Severity Issues

L-01 MIN_FINALITY_THRESHOLD is not set correctly		
Severity: **Low**	Impact: **Low**	Likelihood: **Medium**
Files: src/FastMCTP.sol	Status: Fixed	

Description: The MIN_FINALITY_THRESHOLD is currently set to 0, which contradicts both documentation and code.

The CCTPv2 documentation specifies that the threshold can have a valid range of [1, 2000] and any value set outside the range will be interpreted as being a Standard Transfer, which is contrary to the purpose of the FastMCTP contract.

The CCTPv2 code however, enforces a minimum finality threshold of 500 and will reject any message below this value.

Currently, the off-chain attestation service IRIS enforces a finality of 1000 for all messages <1000, so there is no impact for the moment. Yet when this measure is lifted, it could have breaking consequences for the contract.

Recommendations: It is recommended to set the MIN_FINALITY_THRESHOLD to 500 and implement a guardian only setMinFinality() function which can be used by the protocol team to adjust the threshold to the final minimum once Circle has resolved the contradiction between code and documentation and have lifted the attestation enforcement.

Customer's response: We'll make `minFinalityThreshold` an input variable that would be specified by the user when calling bridge() or createOrder().
Fixed in commit 547847

Fix Review: This resolves the issue. It's worth keeping track of any changes or updates in Circle's docs and informing the users the right value to choose accordingly.

L-02 Guardian can replace the fee manager to steal the redeem fee

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: src/FastMCTP.sol	Status: Acknowledged	

Description: The `redeem()` function sends the redeem fee to the fee manager and then credits the deposit to the redeemer by calling `feeManager.depositFee()`.
A malicious guardian can replace the fee manager with a malicious contract that wouldn't credit those funds to the redeemer and steal those funds instead, causing a loss of funds for the redeemer.

```C/C++
	function depositRelayerFee(address relayer, address token, uint256 amount) internal {
		IERC20(token).transfer(address(feeManager), amount);
		try feeManager.depositFee(relayer, token, amount) {} catch {}
	}
```

Recommendations: Send the funds directly to the redeemer, or put the guardian behind a timelock.

Customer's response: Acknowledged, this is by design. Relayers trust the guardians in this sense.

Informational Severity Issues

I-01. Incorrect revert reason handling in low-level call

Description:
In the `fulfillOrder()` function, when executing a swap through a third-party protocol, the contract incorrectly handles error messages from failed calls. The current implementation attempts to convert the raw return data directly to a string, which will not correctly extract the revert reason from most smart contract calls.

```
C/C++
(bool swapSuccess, bytes memory swapReturn) = swapProtocol.call{value: 0}(swapData);
require(swapSuccess, string(swapReturn));
```

When a smart contract reverts with a reason, the error data is ABI-encoded and follows a specific format that needs to be properly decoded. The current implementation would revert while trying to cast to a string.

Recommendation: Use assembly to properly extract and propagate the revert reason:

```
C/C++
(bool swapSuccess, bytes memory swapReturn) = swapProtocol.call{value: 0}(swapData);
- require(swapSuccess, string(swapReturn));
+ if (!swapSuccess) {
+     assembly {
+         // Revert with the same reason
+         let returnDataSize := mload(swapReturn)
+         revert(add(32, swapReturn), returnDataSize)
+     }
+ }
```

This change ensures that the original revert reason from the swap protocol is correctly propagated back to the caller, which is important for debugging and error handling.

Customer's response: Acknowledged

I-02. Deprecated safe approval method used in approval pattern

Description: The current implementation uses a deprecated `safeApprove` method in the approval pattern within the `approveIfNeeded()` function. This method is no longer recommended by OpenZeppelin due to potential issues with certain ERC20 token implementations.

```
C/C++
function approveIfNeeded(address tokenAddr, address spender, uint256 amount, bool max)
internal {
    IERC20 token = IERC20(tokenAddr);
    uint256 currentAllowance = token.allowance(address(this), spender);

    if (currentAllowance < amount) {
        if (currentAllowance > 0) {
            token.safeApprove(spender, 0);
        }
        token.safeApprove(spender, max ? type(uint256).max : amount);
    }
}
```

The updated SafeERC20 library from OpenZeppelin has replaced `safeApprove` with `forceApprove` and provides better methods for managing allowances, such as `safeIncreaseAllowance` and `safeDecreaseAllowance`.

Recommendation: Update the OpenZeppelin dependencies to the latest version and replace the current approval pattern with the recommended approach:

```
C/C++
function approveIfNeeded(address tokenAddr, address spender, uint256 amount, bool max)
internal {
    IERC20 token = IERC20(tokenAddr);
    uint256 currentAllowance = token.allowance(address(this), spender);

    if (currentAllowance < amount) {
-       if (currentAllowance > 0) {
-           token.safeApprove(spender, 0);
-       }
-       token.safeApprove(spender, max ? type(uint256).max : amount);
+        token.forceApprove(spender, max ? type(uint256).max : amount);
    }
}
```

Alternatively, if you want to only increase the allowance by the necessary amount:

```
C/C++
function approveIfNeeded(address tokenAddr, address spender, uint256 amount, bool max)
internal {
    IERC20 token = IERC20(tokenAddr);
    uint256 currentAllowance = token.allowance(address(this), spender);

    if (currentAllowance < amount) {
-       if (currentAllowance > 0) {
-           token.safeApprove(spender, 0);
-       }
-       token.safeApprove(spender, max ? type(uint256).max : amount);
+       if (max) {
+           token.forceApprove(spender, type(uint256).max);
+       } else {
+           token.safeIncreaseAllowance(spender, amount - currentAllowance);
+       }
    }
}
```

Customer's response: Acknowledged.

I-03. Origin of cross-chain messages is not validated

Description: The FastMCTP protocol validates that messages received are intended for the FastMCTP contract on the destination chain by checking the message recipient, but it does not verify that messages originated from the FastMCTP contract on the source chain. This means that arbitrary messages can be submitted directly by calling TokenMessenger with custom data and then processed on the destination chain through FastMCTP.

While this does not immediately create security issues in the current implementation (as there is limited incentive to bypass the FastMCTP entry point, no user funds are held, and CCTP ensures tokens cannot be minted without equivalent burns), it could lead to unexpected behavior if integrating protocols rely on the integrity of the `customPayload` parameter.
The issue exists because the message validation focuses only on the destination address without checking the source:

```
C/C++
modifier checkRecipient(bytes memory cctpMsg) {
    if (truncateAddress(cctpMsg.toBytes32(CCTPV2_MINT_RECIPIENT_INDEX)) != address(this)) {
        revert InvalidMintRecipient();
    }
    _;
}
```

This allows bypassing the `whenNotPaused` modifier and initial validations that are present in the entry point functions of FastMCTP on the source chain.

Recommendation: Consider adding source validation to ensure messages are sent only from trusted FastMCTP contracts across chains.
Alternatively, clearly document in the protocol specification that the protocol does not validate a message's origin, and that integrating protocols should be aware of these trust assumptions when using `customPayload`.

Customer's response: Acknowledged.

I-04. Potential gas manipulation in eth transfers due to 1/64th rule

Description: The FastMCTP contract contains a potential vulnerability related to the Ethereum 1/64th gas rule when transferring ETH to recipients. The `payEth()` function performs a low-level call to transfer ETH, but it does not require sufficient gas to ensure the transfer succeeds under all circumstances.

```
C/C++
function payEth(address to, uint256 amount, bool revertOnFailure) internal {
    (bool success, ) = payable(to).call{value: amount}('');
    if (revertOnFailure) {
        if (success != true) {
            revert EthTransferFailed();
        }
    }
}
```

Due to Ethereum's 63/64 gas forwarding rule, an attacker might manipulate the gas limit to cause the inner call to fail while allowing the outer transaction to succeed. This would result in ETH being stuck in the contract rather than being forwarded to the intended recipient.

For this to occur:

1. The `payEth()` call must consume a non-negligible amount of gas
2. There must be enough remaining gas for the outer function to complete after the failed ETH transfer
3. The attack is only viable when `inner_call_gas > after_call_gas * 63`

While this issue is unlikely to manifest in practice due to these specific conditions, it represents a theoretical vulnerability that could affect functions like `redeem()`, `fulfillOrder()`, and `refund()` which transfer ETH to users.

Recommendation: Consider implementing one or more of the following measures:

1. Document this limitation in the contract comments to make integrators aware of this edge case
2. Add event emission when ETH transfers fail in non-reverting cases:

```
C/C++
function payEth(address to, uint256 amount, bool revertOnFailure) internal {
    (bool success, ) = payable(to).call{value: amount}('');
    if (!success) {
        if (revertOnFailure) {
            revert EthTransferFailed();
        }
+       emit EthTransferFailed(to, amount);
    }
}
```

Customer's response: Acknowledged.

I-05. Add events for guardian-controlled functions

Description: The following functions don't emit an event:

- setMintRecipient
- setDomainCallers
- setWhitelistedSwapProtocols
- setWhitelistedMsgSenders
- setFeeManager
- setPause
- changeGuardian
- claimGuardian

Emitting an event would allow users to more easily track important changes to the state of the contract.

Recommendation: Emit an event for those functions.

Customer's response: Acknowledged.

I-06. depositRelayerFee should not be called when redeemFee = 0

Description: In redeem(), depositRelayerFee is called regardless whether or not redeemFee > 0. Since with payloadType 2 it is extremely likely that the redeemFee = 0, this amounts to a zero-value call.

Recommendation: Only call depositRelayerFee when redeemFee != 0

Customer's response: Acknowledged.

I-07. Implement Time-Delay for Guardian Functions

Description: If the Guardian were ever to be compromised, it is possible for an attacker to add malicious swap protocols to the whitelist, change the FeeManager, drain all balances from the contract, etc..

While many of these attack vectors can be mitigated, it would be prudent to implement a time-delay on these critical functions. This would allow the protocol to inform users of the incident, which would give them time to evacuate funds.

Recommendation: Implement Time-Delay for critical guardian functions.

Customer's response: Acknowledged.

I-08. Typos in variable naming

Description: There are a few typos in variable names.

The uint256 internal constant CCTPV2_DETINATION_CALLER_INDEX:

- CCTPV2_DE**S**TINATION_CALLER_INDEX

In getMintRecipient:

- mintRec**i**pient instead of mintRec**e**pient.

Recommendation:

Customer's response: Acknowledged.

I-09. Incomplete Events: OrderFulfilled & OrderRefunded

Description: The `OrderFulfilled` and `OrderRefunded` events in the FastMCTP contract emit incomplete information for proper tracking of orders. Currently, these events only include:

```
C/C++
event OrderFulfilled(uint32 sourceDomain, bytes32 sourceNonce, uint256 amount);
event OrderRefunded(uint32 sourceDomain, bytes32 sourceNonce, uint256 amount);
```

The events include the source domain, nonce, and amount, but are missing important information such as the destination address (`destAddr`) and token being transferred (`tokenOut`). This limitation makes it challenging for off-chain systems to accurately track order fulfillment and refunds and for users to associate events with their specific orders.
Recommendation: Improve the event declarations to include the additional relevant parameters:

```
C/C++
- event OrderFulfilled(uint32 sourceDomain, bytes32 sourceNonce, uint256 amount);
+ event OrderFulfilled(uint32 sourceDomain, bytes32 sourceNonce, uint256 amount, address
destAddr, address tokenOut);

- event OrderRefunded(uint32 sourceDomain, bytes32 sourceNonce, uint256 amount);
+ event OrderRefunded(uint32 sourceDomain, bytes32 sourceNonce, uint256 amount, address
destAddr);
```

Customer's response: Acknowledged.

I-10. Unused Errors

Description: The FastMCTP contract includes two unused error declarations that are never thrown throughout the codebase:

```C/C++
error InvalidPayload();
error InvalidAddress();
```

Recommendation: Remove the unused error declarations to improve code clarity.

Customer's response: Acknowledged.

I-11 User can use the `refund()` function to avoid paying protocol fees

Description: When a swap order expires (passes the deadline) the `refund()` function is called to send the user the original USDC amount without swapping. The function doesn't charge any protocol fees in that case.

A user can take advantage of that and bridge funds by creating an order with an extremely short deadline, this would bridge the funds just like the `bridge()/redeem()` functions but without charging any protocol fees.

Recommendations: Consider charging a protocol fee for refunds as well.

Customer's response: Acknowledged.

I-12 User can use redeem fee to avoid paying protocol fees

Description: It is possible for a user to avoid all fees by setting:

- payloadType = 2
- redeemFee = amountIn - circleMaxFee

When redeem() is called, the `redeemFee` is subtracted from the amount, which will leave zero or a dust amount for the protocol and referrer.

```
C/C++
amount = amount - uint256(bridgePayload.redeemFee);
```

Recommendations: Either set a hard cap on redeemFee at f.e. 50% of `amountIn` or apply the fees before subtracting the redeemFee.

Customer's response: Acknowledged.

I-13 Hardcoded eth decimals limit cross-chain compatibility

Description: The FastMCTP contract uses a hardcoded value for ETH decimals (18) which may cause issues when deploying on non-Ethereum chains where the native token has a different number of decimal places.

```
C/C++

uint8 internal constant ETH_DECIMALS = 18;
```

This constant is used in the deNormalizeAmount() function which adjusts token amounts across chains:

```
C/C++
function deNormalizeAmount(uint256 amount, uint8 decimals) internal pure returns(uint256) {
    if (decimals > 8) {
        amount *= 10 ** (decimals - 8);
    }
    return amount;
}
```

The hardcoded value is problematic on chains like Solana (9 decimals), or other networks where the native token might use a different decimal place standard.

Recommendations: When deploying to chains with non-18-decimal native currency, replace the hardcoded constant with a configurable parameter that represents the correct decimal places for the native token on the deployed chain:

```
C/C++

- uint8 internal constant ETH_DECIMALS = 18;
+ uint8 public nativeTokenDecimals;

  constructor(
      address _cctpTokenMessengerV2,
      address _feeManager
+     uint8 _nativeTokenDecimals
```

```
  ) {
      cctpTokenMessengerV2 = ITokenMessengerV2(_cctpTokenMessengerV2);
      feeManager = IFeeManager(_feeManager);
      guardian = msg.sender;
+     nativeTokenDecimals = _nativeTokenDecimals;
  }
```

Customer's response: Acknowledged.

I-14 Common user errors might lead to funds stuck on chain

Description: The following parameters aren't checked during bridging/order creation:

- Gas drop:
  - An unreasonably high amount might be impossible to supply
- Deadline (for orders)
  - An unreasonably high value followed by a price drop might lead to the order getting stuck forever
- Dest address
  - Check it's not set for the zero address

Recommendations: Add sanity checks for those parameters. Alternatively, for gas drop, consider allowing the intended recipient to execute the redemption without providing the gas drop:

```
C/C++
if (bridgePayload.gasDrop > 0) {
    uint256 denormalizedGasDrop = deNormalizeAmount(bridgePayload.gasDrop, ETH_DECIMALS);
-   if (msg.value != denormalizedGasDrop) {
+   if (msg.sender != recipient && msg.value != denormalizedGasDrop) {
        revert InvalidGasDrop();
    }
-   payEth(recipient, denormalizedGasDrop, false);
+   if (msg.value > 0) {
+       payEth(recipient, msg.value, false);
+   }
}
```

Customer's response: Acknowledged.

I-15 Refund event amount includes refund fee

Description: In the refund() function, an event OrderRefunded is emitted which includes the amount refunded. However, the amount does not take into account the `orderPayload.refundFee`, which causes the amount emitted to be slightly larger than the actual amount transferred.

```C/C++
emit OrderRefunded(cctpMsg.toUint32(CCTPV2_SOURCE_DOMAIN_INDEX),
cctpMsg.toBytes32(CCTPV2_NONCE_INDEX), amount);
```

Recommendations: Change the emission to `amount - orderPayload.refundFee`

Customer's response: Acknowledged.

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.