// HALBORN

C3 - CCTP Hub

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **November 13th, 2023 - November 22nd, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	11/21/2023
0.2	Document Updates	11/22/2023
0.3	Draft Review	11/23/2023
0.4	Draft Review	11/23/2023
1.0	Remediation Plan	12/12/2023
1.1	Remediation Plan Review	12/15/2023
1.2	Remediation Plan Review	12/17/2023

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

C3 engaged Halborn to conduct a security assessment on their smart contracts beginning on November 13th, 2023 and ending on November 22nd, 2023. The security assessment was scoped to the smart contracts provided to the Halborn team.

1.2 ASSESSMENT SUMMARY

The team at Halborn was provided nine days for the engagement and assigned a full-time security engineer to verify the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some security risks that were mostly addressed by the C3 team.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions. (solgraph)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs. (MythX)
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Testnet deployment. (Brownie, Anvil, Foundry)

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 EXPLOITABILITY

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 IMPACT

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 SEVERITY COEFFICIENT

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

2.4 SCOPE

IN-SCOPE CODE & COMMITS:

- Repository: c3exchange/c3
  - Commit IDs:
    - 379fb96443961e4cf109cf78592007093ee46fd8
    - 965e2ef7f87225bc1b0fc663910b7a8cd15d75c2
  - Smart contracts **in scope**:
    - app/packages/cctp-support/contracts/C3-CCTP-Hub.sol

OUT-OF-SCOPE:

- Third-party libraries and dependencies.
- Economic attacks.

REMEDIATION COMMIT IDs:

- 965e2ef7f87225bc1b0fc663910b7a8cd15d75c2
- 22fe8217da6aa0d46c61052657b480a5acaef307
- 60673d1e418e9265b3c5b00da59ad9214efcd589

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	0	0	1	3

EXECUTIVE OVERVIEW

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) DEPOSITED TOKENS IN CIRCLE BRIDGE CAN BE PERMANENTLY BLOCKED	Critical (10)	SOLVED - 11/20/2023
(HAL-02) ADDITIONAL PARAMETER IS LOGGED BUT NOT VERIFIED	Low (2.5)	SOLVED - 12/12/2023
(HAL-03) FLOATING PRAGMA	Informational (0.0)	SOLVED - 12/12/2023
(HAL-04) USAGE OF CUSTOM ERRORS INSTEAD OF STRINGS	Informational (0.0)	ACKNOWLEDGED
(HAL-05) UNUSED/DEBUG IMPORTS	Informational (0.0)	SOLVED - 12/12/2023

FINDINGS & TECH DETAILS

4.1 (HAL-01) DEPOSITED TOKENS IN CIRCLE BRIDGE CAN BE PERMANENTLY BLOCKED - CRITICAL(10)

Description:

The burnForWithdraw function acts as a bridge between Wormhole and CircleBridge in order to transfer USDC tokens from C3 protocol in Algorand to any other available chain. This function takes a Wormhole VAA as argument and transfers its bridged tokens to the contract.

When all tokens from Wormhole are transferred to the contract, it takes the msg.sender of the current transaction and uses it as destinationCaller argument in depositForBurnWithCaller function. This argument is used to specify an account that will be the only allowed to mint the bridged tokens on the other side of the CircleBridge as it is specified in the following code in src/MessageTransmitter.sol:

Listing 1: src/MessageTransmitter.sol

```
271 // Validate destination caller
272 if (_msg._destinationCaller() != bytes32(0)) {
273     require(
274         _msg._destinationCaller() ==
275             Message.addressToBytes32(msg.sender),
276         "Invalid caller for message"
277     );
278 }
```

Since msg.sender is being used to define the destinationCaller, this function is prone to front-run attacks, due to the lack of access control to this function, a malicious actor could front-run a legitimate transaction in order to act as destinationCaller during the execution of depositForBurnWithCaller function, blocking indefinitely the transferred tokens in the other side Circle bridge, since the attacker's account will be the only allowed to claim those tokens.

Code Location:

Listing 2: app/packages/cctp-support/contracts/C3-CCTP-Hub.sol

```
142 bytes32 destinationCaller = bytes32(
143     uint256(uint160(address(msg.sender)))
144 );
145
146 nonce =
147     _iCircleBridge.depositForBurnWithCaller(
148         rv.amount,
149         _iCircleIntegration.getDomainFromChainId(chainId),
150         mintRecipient,
151         usdcToken,
152         destinationCaller
153     );
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:H/D:C/Y:N/R:N/S:U (10)

Proof of Concept:

In order to exploit the issue, an attacker just has to follow the next steps:

1. An attacker starts monitoring the mempool in order to find victims trying to execute this function.
2. Once the attacker has identified a valid candidate, the original transaction gets front-run by the attacker's transaction using the same VAA defined as argument in the victim's transaction.
3. Since the attacker managed to front-run the original transaction, the attacker's address will be used as destinationCaller, therefore its account will be the only one allowed to claim all transferred tokens in the other side of the bridge.
4. The attacker does not perform any action at this point, this would mean that all transferred tokens would be blocked in the other side of the bridge.

Recommendation:

It is recommended to use depositForBurn function instead of depositForBurnWithCaller to allow any caller from the other side of the bridge to claim the transferred tokens. In case it's necessary to use depositForBurnWithCaller function, the access to burnForWithdraw function must be restricted to be used by authorized accounts.

Remediation Plan:

SOLVED: The C3 Team solved the issue by removing the destinationCaller code and changing depositForBurnWithCaller with depositForBurn function, so anyone can issue minting.

Commit ID: 965e2ef7f87225bc1b0fc663910b7a8cd15d75c2

4.2 (HAL-02) ADDITIONAL PARAMETER IS LOGGED BUT NOT VERIFIED - LOW (2.5)

Description:

In the redeemAndTriggerDeposit function, it is been identified an argument (initialTxHash) that is not being used other than for emitting an RedeemAndTriggerDepositReturn event. The following description of this argument defines the purpose of it:

> The TX Hash that corresponds to the transfer from the user wallet to the Wormhole Integration Contract. This is logged in this call to be used to track transaction flow for clients such as the C3 Relayer.

Since this parameter is used directly to track information off-chain, it is recommended to perform some sanity checks in order to verify whether the information passed through this argument is correct or not.

Code Location:

Listing 3: app/packages/cctp-support/contracts/C3-CCTP-Hub.sol (Lines 81,115)

```
80 function redeemAndTriggerDeposit(
81     bytes32 initialTxHash,
82     ICircleIntegration.RedeemParameters memory redeemParameters
83 ) external {
84     // redeem our tokens.
85
86     ICircleIntegration.DepositWithPayload
87         memory depositInfo = _iCircleIntegration.
↳ redeemTokensWithPayload(
88             redeemParameters
89         );
90
91     // check the payload now that we have the decoded info,
```

```
↳ validate and revert if
92      // anything is wrong.
93
94      validateC3DepositPayload(depositInfo.payload);
95
96      // Approve the Token Bridge to transfer our freshly minted
↳ tokens.
97
98      address usdcToken = address(uint160(uint256(depositInfo.token)
↳ ));
99
100     IERC20(usdcToken).approve(address(_iTokenBridge), depositInfo.
↳ amount);
101
102     // go ...
103
104     uint64 sequence = _iTokenBridge.transferTokensWithPayload(
105         usdcToken,
106         depositInfo.amount,
107         CHAIN_ID_ALGORAND,
108         bytes32(uint256(uint64(_authorizedC3AppId))),
109         uint32(depositInfo.nonce),
110         depositInfo.payload
111     );
112
113     // keep reference data in a log.
114
115     emit RedeemAndTriggerDepositReturn(initialTxHash, sequence);
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (2.5)

Recommendation:

As it was described, it is recommended, at least, to perform some basic sanity checks in order to verify that the information passed through the aforementioned argument is correct.

Remediation Plan:

SOLVED: The C3 Team solved the issue by adding a check to verify whether the initialTxHash is empty or not.

Commit ID: 22fe8217da6aa0d46c61052657b480a5acaef307

4.3 (HAL-03) FLOATING PRAGMA - INFORMATIONAL (0.0)

Description:

Smart contracts in C3_CCTP_Hub use the floating pragma ^0.8.20. Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, either an outdated compiler version that might introduce bugs that affect the contract system negatively or a pragma version too new which has not been extensively tested.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

Consider locking the pragma version with known bugs for the compiler version by removing the caret (^) symbol. When possible, do not use floating pragma in the final live deployment. Specifying a fixed compiler version ensures that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Remediation Plan:

SOLVED: The C3 Team solved the issue by using a fixed pragma version.

Commit ID: 60673d1e418e9265b3c5b00da59ad9214efcd589

4.4 (HAL-04) USAGE OF CUSTOM ERRORS INSTEAD OF STRINGS - INFORMATIONAL (0.0)

Description:

Failed operations in this contract are reverted with an accompanying message selected from a set of hard-coded strings.

In EVM, emitting a hard-coded string in an error message costs ~50 more gas than emitting a custom error. Additionally, hard-coded strings increase the gas required to deploy the contract.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

Custom errors are available from Solidity version 0.8.4 up. Consider replacing all revert strings with custom errors. Usage of custom errors should look like this:

Listing 4

```
error CustomError();

// ...

if (condition)
    revert CustomError();
```

Remediation Plan:

ACKNOWLEDGED: The C3 Team acknowledged the issue.

4.5 (HAL-05) UNUSED/DEBUG IMPORTS - INFORMATIONAL (0.0)

Description:

It was identified an unnecessary import library in C3_CCTP_Hub used to print values in Hardhat. Since this library is used for debug purposes, it should be removed.

Code Location:

Listing 5: app/packages/cctp-support/contracts/C3-CCTP-Hub.sol

```
8 import "hardhat/console.sol";
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

The aforementioned import should be removed from the smart contract.

Remediation Plan:

SOLVED: The C3 Team solved the issue by removing the aforementioned import.

Commit ID: 60673d1e418e9265b3c5b00da59ad9214efcd589

AUTOMATED TESTING

5.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their ABIs and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

Results:

```
MockTokenBridge._seq (contracts/mocks/MockTokenBridge.sol#47) is never initialized. It is used in:
        - MockTokenBridge.transferTokensWithPayload(address,uint256,uint16,bytes32,uint32,bytes) (contracts/mocks/MockTokenBridge.sol#49-58)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-state-variables

Contract locking ether found:
        Contract MockTokenBridge (contracts/mocks/MockTokenBridge.sol#46-171) has payable functions:
         - MockTokenBridge.transferTokensWithPayload(address,uint256,uint16,bytes32,uint32,bytes) (contracts/mocks/MockTokenBridge.sol#49-58)
        But does not have a function to withdraw the ether
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#contracts-that-lock-ether

C3_CCTP_Hub.redeemAndTriggerDeposit(bytes32,ICircleIntegration.RedeemParameters) (contracts/C3-CCTP-Hub.sol#80-116) ignores return value by IERC20(usdcToken).approve(address(_iTokenBridge),depositInfo.amount) (con
tracts/C3-CCTP-Hub.sol#100)
C3_CCTP_Hub.burnForWithdraw(bytes) (contracts/C3-CCTP-Hub.sol#123-149) ignores return value by IERC20(usdcToken).approve(address(_iCircleBridge),rv.amount) (contracts/C3-CCTP-Hub.sol#138)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return

IUSDC.allowance(address,address).owner (contracts/interfaces/circle/IUSDC.sol#15) shadows:
        - IUSDC.owner() (contracts/interfaces/circle/IUSDC.sol#8) (function)
MockUSDC.constructor(string,string,uint256).name (contracts/mocks/MockUSDC.sol#8) shadows:
        - ERC20.name() (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#58-60) (function)
        - IERC20Metadata.name() (node_modules/@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#15) (function)
MockUSDC.constructor(string,string,uint256).symbol (contracts/mocks/MockUSDC.sol#9) shadows:
        - ERC20.symbol() (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#66-68) (function)
        - IERC20Metadata.symbol() (node_modules/@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#20) (function)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing

Reentrancy in C3_CCTP_Hub.burnForWithdraw(bytes) (contracts/C3-CCTP-Hub.sol#123-149):
        External calls:
        - rv = _iTokenBridge.parseTransferWithPayload(_iTokenBridge.completeTransferWithPayload(vaa)) (contracts/C3-CCTP-Hub.sol#127-130)
        - IERC20(usdcToken).approve(address(_iCircleBridge),rv.amount) (contracts/C3-CCTP-Hub.sol#138)
        - nonce = _iCircleBridge.depositForBurn(rv.amount, _iCircleIntegration.getDomainFromChainId(chainId),mintRecipient,usdcToken) (contracts/C3-CCTP-Hub.sol#140-146)
        Event emitted after the call(s):
        - BurnForWithdraw(_authorizedC3AppId,nonce) (contracts/C3-CCTP-Hub.sol#148)
Reentrancy in C3_CCTP_Hub.redeemAndTriggerDeposit(bytes32,ICircleIntegration.RedeemParameters) (contracts/C3-CCTP-Hub.sol#80-116):
        External calls:
        - depositInfo = _iCircleIntegration.redeemTokensWithPayload(redeemParameters) (contracts/C3-CCTP-Hub.sol#86-89)
        - IERC20(usdcToken).approve(address(_iTokenBridge),depositInfo.amount) (contracts/C3-CCTP-Hub.sol#100)
        - sequence = _iTokenBridge.transferTokensWithPayload(usdcToken,depositInfo.amount,CHAIN_ID_ALGORAND,bytes32(uint256(uint64(_authorizedC3AppId))),uint32(depositInfo.nonce),depositInfo.payload) (contracts/C3
-CCTP-Hub.sol#104-111)
        Event emitted after the call(s):
        - RedeemAndTriggerDepositReturn(initialTxHash,sequence) (contracts/C3-CCTP-Hub.sol#115)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3

Different versions of Solidity are used:
        - Version used: ['>=0.4.22<0.9.0', '>=0.8.0<0.9.0', '^0.8.0', '^0.8.19', '^0.8.20']
        - >=0.4.22<0.9.0 (node_modules/hardhat/console.sol#2)
        - >=0.8.0<0.9.0 (node_modules/solidity-bytes-utils/contracts/BytesLib.sol#9)
        - ^0.8.0 (contracts/interfaces/ITokenBridge.sol#4)
        - ^0.8.0 (contracts/interfaces/IWETH.sol#4)
        - ^0.8.0 (contracts/interfaces/IWormhole.sol#4)
        - ^0.8.19 (contracts/interfaces/ICircleIntegration.sol#3)
        - ^0.8.19 (contracts/interfaces/circle/ICircleBridge.sol#2)
        - ^0.8.19 (contracts/interfaces/circle/IMessageTransmitter.sol#2)
        - ^0.8.19 (contracts/interfaces/circle/ITokenMinter.sol#2)
        - ^0.8.19 (contracts/interfaces/circle/IUSDC.sol#2)
        - ^0.8.19 (contracts/mocks/MockUSDC.sol#2)
        - ^0.8.20 (node_modules/@openzeppelin/contracts/interfaces/draft-IERC6093.sol#3)
        - ^0.8.20 (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#4)
        - ^0.8.20 (node_modules/@openzeppelin/contracts/token/ERC20/IERC20.sol#4)
        - ^0.8.20 (node_modules/@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#4)
        - ^0.8.20 (node_modules/@openzeppelin/contracts/utils/Context.sol#4)
        - ^0.8.20 (contracts/C3-CCTP-Hub.sol#2)
        - ^0.8.20 (contracts/mocks/MockCircleBridge.sol#2)
        - ^0.8.20 (contracts/mocks/MockCircleIntegration.sol#2)
        - ^0.8.20 (contracts/mocks/MockTokenBridge.sol#2)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used

Pragma version^0.8.20 (contracts/C3-CCTP-Hub.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6/0.8.16
Pragma version^0.8.19 (contracts/interfaces/ICircleIntegration.sol#3) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6/0.8.16
Pragma version^0.8.0 (contracts/interfaces/ITokenBridge.sol#4) allows old versions
Pragma version^0.8.0 (contracts/interfaces/IWETH.sol#4) allows old versions
Pragma version^0.8.0 (contracts/interfaces/IWormhole.sol#4) allows old versions
Pragma version^0.8.19 (contracts/interfaces/circle/ICircleBridge.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6/0.8.16
Pragma version^0.8.19 (contracts/interfaces/circle/IMessageTransmitter.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6/0.8.16
Pragma version^0.8.19 (contracts/interfaces/circle/ITokenMinter.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6/0.8.16
Pragma version^0.8.19 (contracts/interfaces/circle/IUSDC.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6/0.8.16
Pragma version^0.8.20 (contracts/mocks/MockCircleBridge.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6/0.8.16
Pragma version^0.8.20 (contracts/mocks/MockCircleIntegration.sol#2) necessitates a version too recent to
be trusted. Consider deploying with 0.6.12/0.7.6/0.8.16
Pragma version^0.8.20 (contracts/mocks/MockTokenBridge.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6/0.8.16
Pragma version^0.8.19 (contracts/mocks/MockUSDC.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6/0.8.16
solc-0.8.20 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity

Contract C3_CCTP_Hub (contracts/C3-CCTP-Hub.sol#34-272) is not in CapWords
Function ITokenBridge._parseTransferCommon(bytes) (contracts/interfaces/ITokenBridge.sol#69) is not in mixedCase
Function ITokenBridge.WETH() (contracts/interfaces/ITokenBridge.sol#131) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions

MockTokenBridge._seq (contracts/mocks/MockTokenBridge.sol#47) should be constant
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-constant

C3_CCTP_Hub._deployer (contracts/C3-CCTP-Hub.sol#43) should be immutable
C3_CCTP_Hub._iCircleBridge (contracts/C3-CCTP-Hub.sol#42) should be immutable
C3_CCTP_Hub._iCircleIntegration (contracts/C3-CCTP-Hub.sol#40) should be immutable
C3_CCTP_Hub._iTokenBridge (contracts/C3-CCTP-Hub.sol#41) should be immutable
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-immutable
. analyzed (22 contracts with 84 detectors), 32 result(s) found
```

- Most of the flagged issues are identified in mock contract; therefore there are no risks since they are not supposed to be deployed in any chain.
- No major issues found by Slither.

THANK YOU FOR CHOOSING

// HALBORN