PUBLIC

Code Assessment

of the Tron-peg USD Coin (USDC) Smart Contracts

October 21, 2024

Produced for

JustCrypto

by

CHAINSECURITY

Contents

1 Executive Summary

Dear JustCrypto Team,

Thank you for trusting us to help JustCrypto with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Tron-peg USD Coin according to Scope to support you in forming an opinion on their security risks.

JustCrypto launches a Tron-peg USDC token on Tron Chain and implements a bridge between Ethereum and Tron to allow users to transfer USDC between the two chains. Users' assets are locked on `TronUSDCBridge` contract controlled by `TronUSDCBridgeController`. An operator of the bridge then mints an appropriate amount of USDC on Tron. Tron USDC is controlled by the `USDCController`.

The most critical subjects covered in our audit are the security of the funds and the liveness and correctness of the bridging process. Only minor issues were uncovered. Security regarding all the aforementioned subjects is high.

The general subjects covered are the efficiency of the implementation, centralization, specification, documentation, and testing. The efficiency of the implementation could be improved in some cases. The centralization of the system is very high. This means that the admins of the system are in full control of the funds on the bridge and Tron. Specification and documentation are sufficient as well as unit testing. End-to-end testing seems to not be sufficient. This is particularly important given that TronVM might differ from Ethereum in unexpected ways.

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

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

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

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

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

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	6
• Code Corrected	5
• Specification Changed	1

2 Assessment Overview

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

2.1 Scope

The assessment was performed on the source code files inside the Tron-peg USD Coin repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	23 Sept 2024	516b1ab54cc3f2ecde00f6344abbe2836a004a86	Initial Version
2	8 Oct 2024	10bee57384cbc7ccf1b30cf0a4ea051af3be0bf5	Fixes
3	14 Oct 2024	9a0396111af3a7ea3d0f9b3665cb408481868592	Add pools on Ethereum

For the solidity smart contracts on Ethereum, the compiler version `0.8.26` was chosen. For the smart contracts on Tron, the compiler version `0.8.20` was chosen.

2.1.1 Included in scope

This report covers the following contracts:

- `contracts/TronUSDCBridge.sol`
- `contracts/TronUSDCBridgeController.sol`
- `contracts/USDC.sol`
- `contracts/USDCController.sol`
- `library/AddressValidator.sol`
- `library/TransactionValidator.sol`

The contracts mentioned above inherit some OpenZeppelin smart contracts and are going to be deployed using Openzeppelin's TransparentUpgradeableProxy upgrade pattern.

More specifically, they inherit:

- `@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol` for TronUSDCBridge, TronUSDCBridgeController, USDC, and USDCController to initialize the contracts.
- `@openzeppelin/contracts-upgradeable/utils/ContextUpgradeable.sol` for TronUSDCBridge, TronUSDCBridgeController, USDC, and USDCController to obtain the contract caller.
- `@openzeppelin/contracts-upgradeable/access/extensions/AccessControlDefaultAdminRulesUpgradeable.sol` for TronUSDCBridgeController and USDCController to do permission control for contracts.
- `@openzeppelin/contracts-upgradeable/utils/ReentrancyGuardUpgradeable.sol` for TronUSDCBridge, TronUSDCBridgeController, USDC, and USDCController to prevent reentrancy attacks.

- `@openzeppelin/contracts-upgradeable/utils/PausableUpgradeable.sol` for TronUSDCBridge, TronUSDCBridgeController, USDC, and USDCController to pause the contracts.
- `@openzeppelin/contracts-upgradeable/access/Ownable2StepUpgradeable.sol` for TronUSDCBridge and USDC for contract owner management.

Additionally, they use:

- `@openzeppelin/contracts/interfaces/IERC20.sol` in TronUSDCBridge, TronUSDCBridgeController, USDC, and USDCController to define the ERC20 interface.
- `@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol` in TronUSDCBridge, TronUSDCBridgeController, USDC, and USDCController for secure ERC20 operations.
- `@openzeppelin/contracts/proxy/transparent/TransparentUpgradeableProxy.sol` as their proxy contract

The version of the aforementioned OpenZeppelin smart contracts is as follows:

- Repository: https://github.com/OpenZeppelin/openzeppelin-contracts
- Branch: release-v5.0
- Commit ID: dbb6104ce834628e473d2173bbc9d47f81a9eec3

Those OpenZeppelin smart contracts have been properly utilized in this project, meeting their business requirements.

2.1.2 Excluded from scope

Any contracts inside the repository that are not mentioned in `Scope` are not part of this assessment. The contracts of the OpenZeppelin library were checked only for their correct usage with the contracts in scope. The current review should not be considered a full security review of these aforementioned dependencies. Tests and deployment scripts are excluded from the scope. The system is assumed to be properly deployed and upgraded. Unless stated otherwise we assume that the semantics of Tron VM are similar to Ethereum. Moreover, we did not consider any potential compiler issues i.e., the compiler producing code with unexpected semantics. The level of decentralization of the system is very low. This means that most of the state can be determined by authorized actors. We assume that these actors will not act maliciously. The deployment and parametrization of the system are assumed to be correct.

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.

JustCrypto offers a bridge between the Ethereum and Tron networks which allows users to transfer USDC. The system consists of two contracts on each network, the bridge and its controller on Ethereum and the USDC and its controller contracts on Tron.

2.2.1 From Ethereum to Tron:

To transfer USDC from Ethereum to Tron, the following steps are necessary:

1. Any user, or any whitelisted user if whitelisting is enabled, calls `TronUSDCBridge.deposit()`, which locks the USDC amount to be bridged to the contract and specifies the receiver address on Tron. There's a minimum amount of USDC that can be bridged.
2. The call emits a `Deposited` event which is caught by an observer.
3. Once the transaction is considered finalized, the mint process is initialized on the Tron chain. Based on the amount, the exact mint process varies:
4. If the bridged amount is smaller or equal to `instantMintThreshold`, the `SYSTEM_OPERATOR_ROLE` can instantly mint it by calling `instantMint()` as long as the amount does not exceed the `instantMintPool` value. Every time a mint is successful this value is further decreased. `instantMint()` does not require any approval.
5. If the bridged amount cannot be instantly minted, a mint request is created by the `SYSTEM_OPERATOR_ROLE` (`requestMint()`). Then the `MINT_RATIFIER_ROLE` parties should approve the amount (`ratifyMint()`). If the amount is smaller or equal to `ratifiedMintThreshold` and there's enough room in the `ratifiedMintPool` only one approval is required. Otherwise, if the amount is smaller or equal to `multiSigMintThreshold` and there's enough room in the `multiSigMintPool` three approvals are required. In any other case, only the `DEFAULT_ADMIN_ROLE` can mint the amount by executing an `ownerMint()`. It is important to note that there's no lower bound for the different minting modes e.g., an amount that could be instantly minted can also go through the ratification process.
6. If there's not enough room for the mint, the minting can be finalized later by anyone by calling `finalizeMint()`.
7. For any amount, during minting a percentage of the amount, defined by the admin, is sent as a fee, defined by `feeRate`, to the treasury.

Minting on Tron is limited by the three pools `instantMintPool`, `ratifiedMintPool` and `multiSigMintPool`. Every time a non-`ownerMint` is successful, the value of the pool is decreased. The pools aim to address issues with users splitting their bridged amounts into smaller portions so that they are below a specific threshold. Any `MINT_RATIFIER_ROLE` can refill the instant pool (`refillInstantMintPool()`). The ratified pool can be refilled by the `DEFAULT_ADMIN_ROLE` or any other `MINT_RATIFIER_ROLE` as long as 2 other ratifiers have approved the refilling (`refillRatifiedMintPool()`). The multiSig pool can only be refilled by the `DEFAULT_ADMIN_ROLE` (`refilleMultiSigMintPool()`). The value required to refill the instant pool should be subtracted from the ratified mint pool and the value required to refill the ratified pool should be subtracted from the multiSig pool. Pools are always fully refilled to a value specified by their respective limits. The `DEFAULT_ADMIN_ROLE` can set these limits. However, such a change does not affect the current value of the pool but only future refills.

The `DEFAULT_ADMIN_ROLE` can use `ownerMint()` to mint USDC on Tron, bypassing the ratification process and the pools.

Once the minting process is finalized, the mint request is deleted from the `mintOperations` list. Therefore, the only on-chain record of the mint request is the events emitted during the minting process. As the `mintOperations` list is ever-growing, the deletion of a request leaves default values in the list which can be used to identify mint requests that have been deleted. However, there is no guarantee that mint request deletion implies that the minting process has been finalized as `revokeMint()` can be called to delete a mint request without finalizing the minting process.

2.2.2 From Tron to Ethereum:

To transfer USDC from Ethereum to Tron, the following steps are necessary:

1. A non-blacklisted and whitelisted user, if whitelisting is enabled, calls `USDC.burn()`, which burns the USDC amount to be bridged. There's a minimum amount of USDC that can be burned.
2. The call emits a `Burned` event which is caught by an observer.
3. Once the transaction is finally confirmed, the withdraw process is initialized on the Ethereum chain. Based on the amount, the exact withdraw process varies:
4. If the bridged amount is smaller or equal to `instantWithdrawThreshold`, the `SYSTEM_OPERATOR_ROLE` can instantly transferred to its receiver (`instantWithdraw()`).
5. If the bridged amount cannot be instantly minted, a withdraw request is created by the `SYSTEM_OPERATOR_ROLE` (`requestWithdraw()`). Then the `WITHDRAW_RATIFIER_ROLE` parties should approve the amount. If the amount is smaller or equal to `ratifiedWithdrawThreshold` only one approval is required (`ratifyWithdraw()`). Otherwise, if the amount is smaller or equal to `multiSigWithdrawThreshold` three approvals are required. In any other case, only the `DEFAULT_ADMIN_ROLE` can mint the amount (`ownerWithdraw()`).
6. If for some reason the withdrawal is not finalized it can be later finalized by anyone by calling `finalizeWithdraw()`.
7. For any amount, during withdrawal, a percentage of the amount, defined by the admin, is sent as a fee, defined by `feeRate`, to the treasury.

The `DEFAULT_ADMIN_ROLE` can use `ownerWithdraw()` to withdraw USDC on Ethereum bypassing the ratification process.

Once the withdrawal process is finalized, the withdraw request is deleted from the `withdrawOperations` list. Therefore, the only on-chain record of the withdraw request is the events emitted during the withdrawal process. As the `withdrawOperations` list is ever-growing, the deletion of a request leaves default values in the list which can be used to identify withdraw requests that have been deleted. However, there is no guarantee that mint request deletion implies that the withdrawal process has been finalized as `revokeWithdraw()` can be called to delete a request without finalizing the withdrawal process.

2.2.3 Ethereum Contracts

2.2.3.1 TronUSDCBridgeController

This proxiable contract is deployed on Ethereum, and it's the owner of the TronUSDCBridge.

During initialization, the `bridge` contract, the `admin`, the `treasury` contract addresses, and the `feeRate` are set (Some of these variables have been removed on Version 2). The `admin` is given the role of `DEFAULT_ADMIN_ROLE` alongside all other defined roles : `SYSTEM_OPERATOR_ROLE`, `MINT_RATIFIER_ROLE`, `ACCESS_MANAGER_ROLE` and `PAUSER_ROLE` and `FUND_MANAGER_ROLE`. The different withdraw thresholds are also set.

Furthermore, it exposes functions for its admins that also change the state of the TronUSDCBridge. In particular:

- `[un]pauseBridge()`: allows the `PAUSER_ROLE` to pause/unpause the bridge.
- `setWhitelistEnabled()` allows the `DEFAULT_ADMIN_ROLE` to enable/disable whitelisting. While whitelisting is enabled only whitelisted users can make use of the functionality of the bridge. Otherwise, any user can as long as they're not blacklisted.
- `updateWhitelist()`: allows the `ACCESS_MANAGER_ROLE` to set the whitelist status for a particular user.

- `setMinDepositAmount()`: allows the `DEFAULT_ADMIN_ROLE` to set a minimum amount of USDC that can be bridged to Tron.
- `setInvestmentAddress()`: allows the `DEFAULT_ADMIN_ROLE` to set the address where an arbitrary amount of tokens stored in the bridge can be sent.
- `transferToInvestmentAddress()`: allows the `FUNDING_MANAGER_ROLE` to transfer a given arbitrary amount of USDC tokens stored in the bridge to the investment address. This aims to use some of the USDC locked in the bridge to earn extra yield for the admins of the system. Note that the contract is assumed to still hold enough assets to facilitate imminent withdrawals making the bridge always solvent.
- `setWithdrawThresholds()`: the `DEFAULT_ADMIN_ROLE` sets the limits used during the bridging of the assets back to Ethereum. These limits determine how many approvals (if any) are needed for a particular amount to be withdrawn to Ethereum.
- `setTreasury()`, `setFeeRate()`, `setBridgeToken()`: the `DEFAULT_ADMIN_ROLE` sets the values.
- `transferBridgeOwnership()`, `claimBridgeOwnership()`: the `DEFAULT_ADMIN_ROLE` transfers ownership to the new owner and the new owner can accept it.
- `revokeWithdraw()`: the `SYSTEM_OPERATOR_ROLE` deletes a pending wtihdraw request. Note that the record of which addresses have given an approval for this request is not cleared up in storage.
- `invalidateAllPendingWithdraws()`: the `DEFAULT_ADMIN_ROLE` invalidates all the withdrawal requests that have been created until the current block is included. It is important to note that it doesn't simply invalidate requests that have existed before the current block, but also those created afterward but still in the same block as with `invalidateAllPendingMints()`.
- `reclaimEtherFromBridge()`, `reclaimEther()` : the `DEFAULT_ADMIN_ROLE` can reclaim respectively any Ether sent to the bridge and to it.
- `reclaimTokenFromBridge()`: the `DEFAULT_ADMIN_ROLE` can reclaim any ERC20 token except for the bridged token sent to the bridge by mistake.
- `reclaimToken()`: the `DEFAULT_ADMIN_ROLE` can reclaim any ERC20 token sent to it.

Note that more functionality has been added in Version 2 of this review.

2.2.3.2 TronUSDCBridge

This proxiable contract is deployed on Ethereum. Its main functionality is to escrow the USDC bridged to Tron. Except for the `deposit()` function which has already been explained, it implements various setters and getters. The setters can all be invoked by the owner of the contract which is the `TronUSDCBridgeController`.

2.2.4 Tron Contracts

2.2.4.1 USDC

`USDC` is the USDC token contract on the Tron network. It is an implementation of the ERC20 standard and aims to be similar to the USDC contract deployed on Ethereum. The contract is upgradable, ownable and pausable. During deployment `initialize()` is called to set the `owner` of the contract. The `owner` can whitelist or blacklist certain addresses.

New tokens can be minted by the `owner` if the contract is not paused and the mint destination address is not blacklisted. In the same way, tokens can be burned by anyone if the contract is not paused and the message sender is not blacklisted and whitelisted if whitelisting is enabled.

Here we present partially the exposed interface of the contract:

- `destroyBlackFunds()`: Blacklisted funds can be burnt by the `owner`. This kind does not take into account the `minBurnAmount`.
- `reclaimToken()` allows the `owner` to reclaim any ERC20 token sent to the contract by mistake.
- `transfer[From]()`: these functions are limited to non-blacklisted receivers and senders. Note in the case of `transferFrom` the `_msgSender()` is not checked to be blacklisted. However, this behavior was changed in the next version of the code base.
- `approve()`: approvals are limited from and to non-blacklisted users.
- `setWhitelistEnabled()` allows to enable/disable whitelisting. While whitelisting is enabled only whitelisted users can make use of the functionality of the bridge and call `burn()`.
- `updateWhitelist()`: allows the `owner` to set the whitelist status for a particular user.

2.2.4.2 USDCController

The `USDCController` is the `owner` of the `USDC` token on Tron. It is upgradable, pausable, and initializable and implements Access Control to restrict access of functionalities to specific roles.

During initialization, the `token` contract, the `admin` and `treasury` contract addresses are set (The initialization of the `treasury` was removed in Version2). The `admin` is given the role of `DEFAULT_ADMIN_ROLE` alongside all other defined roles : `SYSTEM_OPERATOR_ROLE`, `MINT_RATIFIER_ROLE`, `ACCESS_MANAGER_ROLE` and `PAUSER_ROLE`.

Except for the bridging functionality already described the contract exposes the following functionality:

- `[un]pauseToken()`: the `PAUSER_ROLE` can pause/unpause the Tron USDC.
- `[un]pauseMint()`: the `SYSTEM_OPERATORS` can pause/unpause an arbitrary mint request index within the bounds of the `MintOperations` array.
- `addBlacklist()`: the `ACCESS_MANAGER_ROLE` can add addresses to the blacklist of USDC.
- `removeBlacklist()`: the `DEFAULT_ADMIN_ROLE` can remove addresses to the blacklist of USDC. Note that `ACCESS_MANAGER_ROLE` can not remove addresses from the blacklist.
- `updateBurnMinAmount()`: the `DEFAULT_ADMIN_ROLE` can set the minimum amount of USDC that can be bridged back to Ethereum.
- `reclaimTokenFromUSDC()`: the `DEFAULT_ADMIN_ROLE` can reclaim any ERC20 token sent to the USDC contract by mistake.
- `reclaimToken()`: the `DEFAULT_ADMIN_ROLE` can specify an arbitrary address to send any ERC20 token sent to the contract by mistake.
- `setTreasury()`, `setFeeRate()`, `setToken()`: the `DEFAULT_ADMIN_ROLE` sets the respective values of the contract.
- `setMintThresholds()`, `setMintLimits()`: the `DEFAULT_ADMIN_ROLE` sets the instant, ratified, and multisig thresholds and limits as long as these are increasing order.
- `revokeMint()`: the `SYSTEM_OPERATORS` can delete a mint request. Note that the record of the ratifiers which have already approved the mint is not cleared up.
- `invalidateAllPendingMints()`: it works in a similar way as with withdraws. The `DEFAULT_ADMIN_ROLE` invalidates all the mint requests that have been created up to the current block.

2.3 Roles and Trust model

We inferred the following trust model for the system. Note that the trust model was slightly modified for **Version 2**.

The owner of USDC on Tron should be the USDCController contract.

USDCController's access control defines the following roles:

- DEFAULT_ADMIN_ROLE : can grant and revoke any role and has all the permissions.
- SYSTEM_OPERATOR_ROLE : handles the mint operations such as creating, revoking or pausing them.
- MINT_RATIFIER_ROLE : can ratify mint operations and refill the instant and ratify mint pools.
- ACCESS_MANAGER_ROLE : can add an address to the blacklist and update the whitelist.
- PAUSER_ROLE : can pause and unpause the USDC and USDCController contract.

The owner of TronUSDCBridge on Tron should be the TronUSDCBridgeController contract.

TronUSDCBridgeController's access control defines the following roles:

- DEFAULT_ADMIN_ROLE : can grant and revoke any role and has all the permissions.
- SYSTEM_OPERATOR_ROLE : handles the withdrawal operations such as creating, revoking or pausing them.
- WITHDRAW_RATIFIER_ROLE : can ratify withdraw operations.
- ACCESS_MANAGER_ROLE : can update the whitelist.
- PAUSER_ROLE : can pause and unpause the TronUSDCBridge and TronUSDCBridgeController contracts.
- FUND_MANAGER_ROLE : can transfer USDC for investment.

The centralized bridging system is assumed to implement correct accounting and transaction consumption. Furthermore, Ethereum gas fees are assumed to be paid by the centralized party for withdrawals.

The DEFAULT_ADMIN_ROLE and SYSTEM_OPERATOR_ROLE roles are fully trusted and expected to not mint more USDC on Tron than the amount of USDC held on Ethereum. Furthermore, the admins of the system are responsible for the liveness of the system such that for example blocked transactions due to blacklisting can be resolved. The SYSTEM_OPERATOR_ROLE is trusted to not submit more than one mint or withdraw request for a corresponding deposit or burn action. However, a "malicious" SYSTEM_OPERATOR_ROLE should *not* be able to mint more USDC than the size of the instant mint pool and withdraw more USDC than the size the instant withdraw pool.

The FUND_MANAGER is trusted to invest the USDC in a secure 3rd party investment contract and is fully trusted with the funds.

The MINT_RATIFIER_ROLE and WITHDRAW_RATIFIER_ROLE are trusted to approve mint and withdraw requests respectively. The MINT_RATIFIER_ROLE is also trusted to not approve more mints than the amount of USDC held on Ethereum such that every minted token on Tron is fully backed.

The system is expected to always be able to let anyone withdraw their USDC from Tron to Ethereum. The WITHDRAW_RATIFIER_ROLE is trusted to approve withdraw requests such that the system can always fulfill the withdrawal requests.

It is assumed that the system will only be deployed with valid USDC addresses. Other contracts implementing arbitrary behaviors are assumed to never be used as bridged assets. Changing the bridged token after the system is deployed can be problematic as users won't be able to bridge back their assets. Note that setting the bridged token after deployment was removed from Version 2.

2.3.1 Changes in VERSION 2

In Version 2 extensive sanity checks were added on the Tron and Ethereum addresses and transaction hashes. Withdrawing and minting to address 0 is also no longer possible.

Furthermore, `acceptDefaultAdminTransfer()` was added both in `USDCController` and `TronUSDCBridgeController`. This function will revoke all the roles from the current `DEFAULT_ADMIN_ROLE`, transfer the `DEFAULT_ADMIN_ROLE` to the pending admin, and finally grant all the roles to the new `DEFAULT_ADMIN_ROLE`.

`USDCController.setMintLimits()` was modified to reduce the size of the pools if the mint limits are set below the current value of the pools.

`setBridgeToken()` in `TroUSDCBridgeController` and `setToken()` in `TronUSDCBridge` were removed. The bridge token is now set during deployment and can't be changed.

`USDC.transferFrom()` has been modified and now also verifies that `_msgSender()` is not blacklisted.

`USDCController.initialize()` no longer initializes the `treasury` value. The `treasury` must now be set with `setTreasury()` by the `DEFAULT_ADMIN_ROLE`.

`reclaimTRX()` and `reclaimTRXFromUSDC()` were added to the `USDCController` which allow to claim native assets from the `USDCController` and the USDC contract respectively.

Finally, the `treasury` is allowed to be set to 0 as long as the fee rates are non-zero.

Version 2 also modified permissions for the following functions:

- `unpauseBridge()` and `unpause()` in `TronUSDCBridgeController` and `USDCController` can now only be called by the `DEFAULT_ADMIN_ROLE`.
- `pauseWithdraw()` in `TronUSDCBridgeController` can now only be called by the `PAUSER_ROLE`.
- `unpauseWithdraw()` in `TronUSDCBridgeController` can now only be called by the `DEFAULT_ADMIN_ROLE`.
- `pauseMint()` in `USDCController` can now only be called by the `PAUSER_ROLE`.
- `unpauseMint()` in `USDCController` can now only be called by the `DEFAULT_ADMIN_ROLE`.

2.3.2 Changes in Version 3

The pool mechanism was added for the withdrawal process. This mechanism is identical to the one on the TRON side when processing mint requests.

3 Limitations and use of report

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

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

4 Terminology

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

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

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

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

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

5 Findings

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

- 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	0

6 Resolved Findings

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

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

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

- Bridging to Zero Address CODECHANGE Code Corrected
- Changing Mint Limits Code Corrected
- Missing Check on USDC.transferFrom Code Corrected
- Missing Sanity Checks Code Corrected
- Tron Native Assets Cannot Be Reclaimed From Contracts on the TRON Network Code Corrected
- Zero Fees on Ethereum Side Can Lead to Gas Griefing Specification Changed

Informational Findings	3

- Disabling Initializers Specification Changed
- Frontrunning Concerns Specification Changed
- Gas Optimizations Code Corrected

6.1 Bridging to Zero Address CODECHANGE

Correctness Low Version 1 Code Corrected

CS-TRUB-006

In `USDC.burn()`, an `amount` and a `targetEthAddress` are required as arguments such that the `amount` is subtracted from the balance of the caller on the Tron network and the equivalent `amount` will be sent to the `targetEthAddress` on Ethereum. However, the `targetEthAddress` is not properly checked to be non-zero. However, `requestWithdraw()` will fail if the provided destination address is 0. This means that a `burn()` can succeed with a 0 address as a destination but its corresponding `requestWithdraw()` will fail. The same holds for `instantWithdraw()`.

Similarly, there is also no check in `deposit()` that `targetTronAddress` is non-zero. However, the Tron destination address is checked to be non-zero in `instantMint()` and `_requestMint()`.

Code corrected:

`USDC.burn()` now checks that `targetEthAddress` is valid and not a zero address, and `TronUSDCBridge.deposit()` now checks that `targetTronAddress` is valid and not a zero address.

6.2 Changing Mint Limits

Design | Low | Version 1 | Code Corrected

CS-TRUB-011

Reducing the mint limits doesn't affect the current value of the pools. Consider the following scenario:

- The `instantMintLimit` is 100,
- The `instantMintPool` gets refilled to 100,
- Then the `instantMintLimit` is set to 10.

In this case the `instantMintPool` is still 100 so it won't be affected by the new value of the `instantMintLimit` even if the current value of the `instantMintPool` exceeds the maximum limit set.

Code corrected:

`USDCController.setMintLimits()` now updates the pools in such cases.

6.3 Missing Check on `USDC.transferFrom`

Design | Low | Version 1 | Code Corrected

CS-TRUB-005

USDC on Tron aims to be the counterpart of USDC on Ethereum. However, USDC does not check if `_msgSender()` is blacklisted in `transferFrom()` contrary to the Ethereum implementation.

Code corrected:

`USDC.transferFrom()` was modified to also check if `_msgSender()` is blacklisted or not.

6.4 Missing Sanity Checks

Correctness | Low | Version 1 | Code Corrected

CS-TRUB-007

We identified the following missing sanity checks:

1. `setMinDepositAmount()` and `updateBurnMinAmount()` do not prevent setting the minimum deposit and burn amount to zero whereas `initialize()` does.
2. `USDCController.initialize()` enforces that:
   - `_ratifiedWithdrawThreshold > _instantWithdrawThreshold` and
   - `_multiSigWithdrawThreshold > _ratifiedWithdrawThreshold`

   However `setMintThresholds()` allows for the three thresholds to be equal. The same issue is present in `TronUSDCBridgeController` between `initialize()` and `setWithdrawThresholds()`.
3. `USDCController.initialize()` enforces that:
   - `_ratifiedMintLimit > _instantMintLimit` and

- `multiSigMintLimit > _ratifiedMintLimit`.

However `setMintThresholds()` allows for the three limits to be equal.

4. `USDCController.initialize()` enforces that:
   - `_instantMintThreshold > 0` and
   - `_instantMintLimit > 0`,

   However `setMintThresholds()` and `setMintLimits()` allow for the instant threshold and limit to be set to zero. The same issue is present in `TronUSDCBridgeController` between `initialize()` and `setWithdrawThresholds()`.
5. `USDCController.initialize()`, `USDCController.setMintLimits()` and `setMintThresholds()` do not enforce that the different pool limits (instant, ratified and multisig) should be greater or equal to their respective thresholds.
6. `ethDepositTx` in `USDCController.instantMint()` and `USDCController._requestMint()` is not checked to not be equal to zero nor verified to be of valid length for an Ethereum transaction hash.
7. When ratifying an operation an arbitrary index is specified. This index is never sanitized. Therefore, an old, already deleted `index`, can be approved.

Code partially corrected:

1. `setMinDepositAmount()` and `updateBurnMinAmount()` now prevent setting the minimum deposit and burn amounts to zero.

4. `USDCController.setMintThresholds()`, `USDCController.setMintLimits()`, and `TronUSDCBridgeController.setWithdrawThresholds()` have respectively added a check that `_instant` is not 0.
5. `USDCController.initialize()`, `USDCController.setMintLimits()` and `setMintThresholds()` now enforces that the different pool limits (instant, ratified and multisig) are greater than or equal to their respective thresholds.
6. `ethDepositTx` in both `USDCController.instantMint()` and `USDCController._requestMint()` is now checked to not be equal to zero and to be of valid length for an Ethereum transaction hash.
7. In `TronUSDCBridgeController.ratifyWithdraw()`, checks have been implemented to ensure that `_to` and `_value` are not null, and `tronBurnTx` is both non-null and a valid Tron transaction hash. Similarly, `USDCController.ratifyMint()` now verifies that `_to` and `_value` are not null, and `ethDepositTx` is both non-null and a valid Ethereum transaction hash.

6.5 Tron Native Assets Cannot Be Reclaimed From Contracts on the TRON Network

Design | Low | Version 1 | Code Corrected

CS-TRUB-004

Native assets sent to `USDCController` and USDC on the Tron network cannot be reclaimed contrary to `TronUSDCBridgeController` and `TronUSDCBridge` where native ETH can be reclaimed.

Code corrected:

The missing functionality has been added to both the `USDC` and `USDCController` contracts. `reclaimTRXFromUSDC()` was also added to `USDCController` to call `reclaimTRX()` on `USDC`.

6.6 Zero Fees on Ethereum Side Can Lead to Gas Griefing

Design | Low | Version 1 | Specification Changed

CS-TRUB-003

When bridging assets, a message is emitted for the target chain which is handled by authorized actors controlled by the bridge admins. These actors pay for the transaction fees and should be compensated by the fees charged. Setting fees to zero, especially on the Ethereum side could create opportunities for a griefing attack. A malicious actor could spam the bridge with many `burn()` transactions. These would trigger many `requestWithdraw()` on the Ethereum which would be paid by the bridge operator. This can be especially dangerous in the case of actors staking big amounts on Tron. These actors are rewarded with energy which they can redeem to pay for their transactions. This means that these stakers can execute transactions for free.

Specification changed

JustCrypto replied: "During the initial phase, only whitelisted users will be permitted to conduct cross-chain transactions, and the minimum cross-chain amount will be set at a higher value, thereby significantly reducing the likelihood of gas griefing. In the later stages, when access restrictions are lifted to allow everyone to engage in cross-chain transactions, we will consider implementing a non-zero fee rate."

6.7 Disabling Initializers

Informational | Version 1 | Specification Changed

CS-TRUB-008

All the contracts will be deployed behind a proxy. The state of these contracts is set by calling `initialize()`. However, the contract storing the implementation logic can also be initialized, allowing anyone to set its state arbitrarily. The initialization process can be blocked during its construction.

Specification change

After the deployment of the logic contract, `initialize()` will be called immediately to ensure that no one else can call it.

6.8 Frontrunning Concerns

Informational | Version 1 | Specification Changed

CS-TRUB-009

Bridging of USDC is subject to front-running issues. For example, the `feeRate` can be increased while a user already initiated the bridging assuming the fee will remain unchanged. The same applies to blacklisting as the destination address of the bridging action can be blacklisted during the bridging

process. Note that these concerns are inherent in actions that don't happen atomically and depend on the ordering of the transactions.

Specification changed

JustCrypto replied: "We will ensure that all cross-chain bridge transactions are completed prior to making any adjustments to the fee rate, and we will provide advance notice to users through official communication channels."

6.9 Gas Optimizations

Informational | Version 1 | Code Corrected

CS-TRUB-010

We identified the following gas optimizations. Note that on Tron storage access is not cached. Therefore, reading the same storage slot repeatedly doesn't cost less gas:

1. `ratifyMint()` and `ratifyWithdraw()` both take as arguments redundant arguments such as `_to`, `_value` and `_tronBurnTx` or respectively `ethDepositTx`. These arguments can all be recovered using the request `_index`.
2. `instantWithdraw()` reads from storage before sanitizing the function arguments.
3. The `WithdrawOperation`, `WithdrawOperationView`, `BridgeStorage`, `MintOperation` and `MintOperationView` structs could all be one slot smaller if the boolean value is moved right after the first field of type `address`.
4. `USDCStorage` struct could use one less slot if `minBurnAmount` was represent on at most 31 bytes instead of 32 such that `whitelistEnabled` could be moved to the same slot as `minBurnAmount`.
5. `treasury` in both `USDCController` and `TronUSDCBridgeController` can never be equal to zero due checks in `initialize()` and `setTreasury()`. However, `instantWithdraw()`, `finalizeWithdraw()`, `instantMint()` and `finalizeMint()` perform additional checks on `treasury` to be different from zero which is redundant.
6. `vaultAddress` in `TronUSDCBridgeController` is never used.
7. `USDC_DECIMALS` in `TronUSDCBridge` is never used.

Code changed

JustCrypto implement to following changes:

1. In version 2 `ratifyMint()` and `ratifyWithdraw()` should no longer be modified as they're needed to make sure that no deleted request will be further ratified.
2. `instantWithdraw()` and `instantMint()` now sanitize the function arguments before reading from storage.
3. The structures `WithdrawOperation`, `WithdrawOperationView`, `BridgeStorage`, `MintOperation`, and `MintOperationView` have been modified by moving the boolean fields after the first address type field to optimize storage slot usage.
4. The type of the `minBurnAmount` field in the `USDCStorage` structure has been changed from `uint` to `uint248`.
5. In `USDCController` and `TronUSDCBridgeController`, `instantWithdraw()`, `finalizeWithdraw()`, `instantMint()`, and `finalizeMint()` cancel the additional checks

on the treasury. And we add logic in `setFeeRate()` method to make sure if a non-zero `feeRate` is set, the treasury must not be address(0).

6. `vaultAddress` has been removed.
7. `USDC_DECIMALS` has been removed.

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 Approvals for Paused or Invalidated Requests

Informational **Version 1** **Acknowledged**

CS-TRUB-001

`USDCController.ratifyMint()` and `TronUSDCBridgeController.ratifyWithdraw()` can be called by ratifiers to approve a pending request. If the request cannot yet be finalized i.e., `hasEnoughApprovals()` returns false, a paused or invalidated request can be successfully approved as there are no further checks. Such a request, however, cannot get approved in case it has enough approvals but `_canFinalize()` reverts. Indeed, in such a case, the transaction will revert and, therefore, the last approval will not be recorded. In other words, whether a request can be approved depends on its state only for the last approval.

```
// USDCController.sol, a similar condition is present in
// TronUSDCBridgeController.sol for withdraw requests
if (hasEnoughApproval(op.numberOfApproval, _value)) {
    finalizeMint(_index);
}
```

Acknowledged:

JustCrypto acknowledges the issue with the following statement:

Ratifiers will first ensure that the request can be finalized before approving it.

7.2 Repetitive Blacklisting

Informational **Version 1** **Acknowledged**

CS-TRUB-002

In `USDC.sol` the `owner` can `blacklist` an account. However, the same account can be blacklisted multiple times, resulting in multiple `BlacklistUpdated` events being emitted for the same account. The same holds for `unblacklist()`.

Similarly, in both `USDC.sol` and `TronUSDCBridge.sol`, `updateWhitelist()` will always emit an event in case of repetitive whitelisting or un-whitelisting.

Pausing, unpausing or invalidating requests also emits redundant events.

Acknowledged:

JustCrypto acknowledges this issue with the following statement:

These functions are low-frequency operations in our business. Even if there is repeated setting, the gas consumption is acceptable.

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 Consuming Transactions Multiple Times

Note Version 1

When a user bridges assets, a message is emitted to the target chain. This message should be consumed by an operation invoked by an authorized user. Note that there are no measures to prevent the consumption of this message multiple times as the consumed transactions are only emitted as events and are not accessible by the runtime. There is no logic preventing a ratifier from submitting the same request or invoking the same instant withdrawal or mint. Extra caution should be taken off-chain to make sure that the ratifiers are aware of instant mints or requests submitted by other ratifiers to avoid consuming the same transaction multiple times.

JustCrypto added the following statement : "Our solution to address these issues is an off-chain reconciliation system. Each operation within the cross-chain process will undergo systematic reconciliation, ensuring the accuracy and consistency of cross-chain transactions."

8.2 Differences Between `minDepositAmount` and `minBurnAmount` Can Lead to Funds Being Stuck

Note Version 1

In a scenario where `minDepositAmount` is 10 and `minBurnAmount` is 20, a user can deposit 10 USDC on Ethereum and mint 10 USDC on Tron. However, the user will not be able to burn the 10 USDC on Tron to withdraw the 10 USDC on Ethereum. This can lead to funds being stuck in the system and can force the user to bridge additional USDC from Ethereum to Tron to be able to withdraw the funds back to Ethereum.

JustCrypto added to following statement : "We will ensure that the `minDepositAmount` and `minBurnAmount` are consistent, thereby preventing potential issues where users' funds might become locked. We will also provide sufficient liquidity on the Tron chain to ensure that users can easily obtain USDC."

8.3 Usage of Tron Solidity Compiler

Note Version 1

Smart contracts or Tron are compiled using a forked version of the solidity compiler. Unexpected behavior might occur due to the fact that Tron forked the solidity compiler and implemented changes that might not have been audited yet. Moreover, the Tron solidity compiler version is not fixed by the project's configuration.

JustCrypto added the following statement : "The Tron network has been running stably for many years, with numerous smart contracts successfully deployed on it. Our project has also undergone thorough testing on both testnet and mainnet, verifying the functionality and reliability of the contracts. We use the latest version of the compiler to ensure the security, performance, and stability of the smart contracts, avoiding known vulnerabilities, improving execution efficiency, and maintaining compatibility with the latest updates on the network"

8.4 Withdraw Can Block Due to USDC Blacklist

Note Version 1

To bridge USDC from Tron to Ethereum, USDC on Tron is burnt and an equivalent amount of USDC is withdrawn to the destination address on Ethereum. However, the withdrawal process can revert if the receiver address is blacklisted by USDC on Ethereum. The same applies to bridging USDC from Ethereum to Tron.

JustCrypto added to following statement : "For users on the blacklist, we will handle their assets according to compliance opinions."