PUBLIC

Code Assessment

of the stUSDT

Smart Contracts

February 26, 2024

Produced for

by

CHAINSECURITY

Contents

1 Executive Summary

Dear stUSDT team,

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

stUSDT is a custodial system for providing off-chain yield to users on-chain. Users receive the stUSDT token as a representation of their deposit in the system and can create a withdrawal request to turn their deposit back into USDT.

The most critical subjects covered in our audit are asset solvency, functional correctness, and access control. Functional correctness is good, but there were some issues uncovered, such as Rounding Errors in TRC20 methods. Security regarding the other subjects is high. Note that any off-chain parts of the system are out of the scope of this review.

The general subjects covered are unit testing, documentation, code complexity, and gas efficiency. Unit testing is non-existent, as no unit tests were provided with the code. Documentation is improvable, as the code is missing NatSpec on many functions, and no public documentation page exists. Code complexity is improvable, as low-level code is used in places where it is not necessary. The proxy pattern used works correctly but does not follow best practices that aid in avoiding mistakes during upgrades. See Proxy Upgrades Must Be Well-tested. Gas efficiency is good.

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

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

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

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

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

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1
• Code Corrected	1
Low-Severity Findings	4
• Code Corrected	3
• Risk Accepted	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 provided by stUSDT, based on the documentation files. No Git repository was provided. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	11 December 2023	a00c9f6f8b13cb727dc54d60a7429b5268e4e2ef	Initial Version
2	15 January 2024	dc22fa6d3c127d52cfb87023d6714a4b6982613b	First fixes
3	1 February 2024	12c225c7bb8a1758b6fc2c04d74cb698796213c6	Additional fixes

For the solidity smart contracts, the compiler version `0.8.18` was chosen.

The following files were in scope of this review:

- UnstUSDTProxy.sol
- WstUSDTStorage.sol
- BlackListManager.sol
- MinterProxy.sol
- StUSDTG1.sol
- AdminProxy.sol
- UnstUSDTStorage.sol
- MinterG1.sol
- WstUSDTG1.sol
- WstUSDTProxy.sol
- StUSDTProxy.sol
- ValuesAggregator.sol
- StUSDTStorage.sol
- WstUSDTG2.sol
- AdminStorage.sol
- MinterStorage.sol
- MinterForTUSD.sol
- AssetSwapRouter.sol
- UnstUSDTG1.sol
- FOR_ETHEREUM_CONTRACT/UnstUSDTProxy.sol
- FOR_ETHEREUM_CONTRACT/WstUSDTStorage.sol
- FOR_ETHEREUM_CONTRACT/BlackListManager.sol
- FOR_ETHEREUM_CONTRACT/MinterProxy.sol

- FOR_ETHEREUM_CONTRACT/StUSDTG1.sol
- FOR_ETHEREUM_CONTRACT/AdminProxy.sol
- FOR_ETHEREUM_CONTRACT/UnstUSDTStorage.sol
- FOR_ETHEREUM_CONTRACT/MinterG1.sol
- FOR_ETHEREUM_CONTRACT/WstUSDTProxy.sol
- FOR_ETHEREUM_CONTRACT/StUSDTProxy.sol
- FOR_ETHEREUM_CONTRACT/ValuesAggregator.sol
- FOR_ETHEREUM_CONTRACT/StUSDTStorage.sol
- FOR_ETHEREUM_CONTRACT/WstUSDTG2.sol
- FOR_ETHEREUM_CONTRACT/AdminStorage.sol
- FOR_ETHEREUM_CONTRACT/MinterStorage.sol
- FOR_ETHEREUM_CONTRACT/UnstUSDTG1.sol
- FOR_ETHEREUM_CONTRACT/interface/IStUSDT.sol
- FOR_ETHEREUM_CONTRACT/interface/IBlackListManager.sol
- FOR_ETHEREUM_CONTRACT/interface/IERC20.sol
- FOR_ETHEREUM_CONTRACT/interface/IProxy.sol
- FOR_ETHEREUM_CONTRACT/library/TransferHelper.sol
- FOR_ETHEREUM_CONTRACT/library/EnumerableSet.sol
- FOR_ETHEREUM_CONTRACT/pauseOperation/PauseController.sol
- FOR_ETHEREUM_CONTRACT/pauseOperation/utils/AccessControl.sol
- FOR_ETHEREUM_CONTRACT/pauseOperation/utils/EnumerableSet.sol
- FOR_ETHEREUM_CONTRACT/pauseOperation/utils/AccessControlMixin.sol
- FOR_ETHEREUM_CONTRACT/pauseOperation/utils/AccessControlSingle.sol
- FOR_ETHEREUM_CONTRACT/pauseOperation/utils/Address.sol
- FOR_ETHEREUM_CONTRACT/pauseOperation/utils/Initializable.sol
- FOR_ETHEREUM_CONTRACT/pauseOperation/utils/Context.sol
- interface/IStUSDT.sol
- interface/IBlackListManager.sol
- interface/IProxy.sol
- interface/ITRC20.sol
- pauseOperation/PauseController.sol

2.1.1 Excluded from scope

Any off-chain components of the system are out of scope.

Library and utils functions were out of scope of this review.

- library/TransferHelper.sol
- library/EnumerableSet.sol
- pauseOperation/utils/AccessControl.sol

- pauseOperation/utils/EnumerableSet.sol
- pauseOperation/utils/AccessControlMixin.sol
- pauseOperation/utils/AccessControlSingle.sol
- pauseOperation/utils/Address.sol
- pauseOperation/utils/Initializable.sol
- pauseOperation/utils/Context.sol

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.

stUSDT is a system that allows holders of USDT (and potentially other stablecoins) to deposit their tokens in exchange for the receipt token stUSDT. The admins can then withdraw the USDT to perform off-chain investments with them, such as buying bonds. A user can make a withdrawal request, which will allow them to exchange their stUSDT back to USDT if the contract has sufficient funds. If not, the user will need to wait until the admins deposit additional funds. The system is fully custodial. The contracts have no way to enforce that admins honor withdrawal requests.

2.2.1 StUSDTG1

stUSDTG1 is the contract implementing the TRC20/ERC20 receipt token stUSDT, that depositors to the system receive. It has Minter and Burner roles, as well as an admin, a mintPausedAdmin and a rebaseAdmin. It also supports a blacklist. Addresses on the blacklist cannot transfer any tokens. There is a cap `maxTotalUnderlying` that limits deposits once the maximum outstanding amount of stUSDT is reached. The cap can be surpassed with rebases.

stUSDT has a rebasing mechanism, which can change the `totalSupply` of the token. Users' balances are tracked internally as shares. When a rebase happens, a user's shares stay the same, but their `balance` increases. The balance is calculated as `userShares * totalSupply / totalShares`. The rebaseAdmin can call the `increaseBase` and `decreaseBase` functions to execute a positive or negative rebase. The maximum rebase amount and frequency can be set by the admin. The rebase is intended to keep the value of one stUSDT equal to one USDT.

stUSDT implements all functions defined in TRC20/ERC20. Additionally, it implements the following most important functions:

1. `transferShares()`: Like `TRC20.transfer()`, but takes an amount of shares instead of tokens.
2. `mint()`: A registered minter can call this to mint an amount of stUSDT to an address. The address must not be on the blacklist.
3. `burnShares()`: A registered burner can call this to burn an amount of shares.
4. `increaseBase()`: The rebaseAdmin can call this to increase the total supply of stUSDT by a stUSDT amount.
5. `decreaseBase()`: The rebaseAdmin can call this to reduce the total supply of stUSDT by a stUSDT amount.
6. `setMintPaused()`: The mintPausedAdmin can call this to enable/disable the minting of additional stUSDT by all minters.

Additionally, there are setters for parameters and roles, that can be called only by the admin role.

Note that stUSDT has 18 decimals, while USDT has only 6 decimals. Also, note that the rebase is uniform. When a rebase happens, it does not matter if a user has been holding for the entire period since the last rebase, or if they acquired their stUSDT just before the rebase. As rebases are expected to be frequent, the effect of this should be minimal (and smaller than the withdrawal fee).

2.2.2 MinterG1

MinterG1 is intended to fill the minter role of stUSDT. It allows users to mint stUSDT by depositing the underlying token (in this case USDT). The financeAdmin role is allowed to withdraw any USDT in the contract to a separate `custody` address, to make off-chain investments with them. The other roles are the admin and mintPausedAdmin.

MinterG1 implements the following most important functions:

1. `submit()`: Takes USDT from the user and mints a corresponding amount of stUSDT to their address. There is a minimum amount that must be submitted, configurable by the admin.
2. `extract()`: The financeAdmin can call this to withdraw any USDT in the contract to the `custody` address.
3. `setStakingPaused()`: The mintPausedAdmin can call this to enable/disable the minting of additional stUSDT using the underlying token of this specific minter.

Additionally, there are setters for parameters and roles, that can be called only by the admin role.

The MinterForTUSD contract is equivalent to MinterG1, except for variable renaming and using the TransferHelper lib for interacting with TRC20/ERC20 tokens. It is intended to be used to allow minting of stUSDT using TUSD as underlying token (instead of USDT).

2.2.3 UnstUSDTG1

UnstUSDTG1 is intended to fill the burner role of stUSDT. It allows users to burn their stUSDT to make a withdrawal request, which should eventually return them an equivalent amount of USDT. There is a withdrawal fee (in percent) and a `minDelayTime`, which can be configured by the admin. There is also a financeAdmin role.

A withdrawal has multiple phases:

1. A user calls `requestWithdrawal()`. This burns their stUSDT immediately and creates a new, non-finalized withdrawalRequest for them.
2. The financeAdmin should make sure that there are sufficient USDT in the UnstUSDT contract to cover all previously finalized withdrawals, plus the new one. If there aren't, the financeAdmin should liquidate off-chain assets and deposit additional USDT to the contract.
3. Once there are sufficient funds in the contract, the financeAdmin calls `finalize()` to finalize the new withdrawalRequest.
4. Once the request is finalized and the `minDelayTime` has passed, the user may call `claimWithdrawals()`. This will transfer a USDT amount equivalent to the burnt stUSDT, minus the withdrawal fee, to the user.

There is a parameter `finalizingPaused`. If this is set to `true`, then all requests made are immediately automatically finalized without needing intervention from the financeAdmin. In this case, the steps 2. and 3. are skipped.

UnstUSDTG1 implements the following most important functions:

1. `requestWithdrawal()`: Burns an amount of stUSDT from the caller and creates a new withdrawalRequest for them. The amount must be greater than the minimum withdrawal amount. Not callable by addresses that are on the blacklist.
2. `requestWithdrawByShares()`: Same as `requestWithdrawal()`, but the amount is specified in stUSDT shares.

3. `claimWithdrawals()`: Takes an array of finalized requestIDs and transfers an amount of USDT equivalent to the users' burnt stUSDT minus withdrawal fees to the caller. The caller must be the owner of the requests.
4. `finalize()`: The financeAdmin can call this to finalize requests. He can specify a `_lastRequestIdToFinalize`. All requests with an id smaller or equal to the given Id will be finalized at once. A `_maxAmountOfToken` of USDT can also be specified. This will ensure that the sum of tokens that are finalized in all newly finalized requests cannot exceed `_maxAmountOfToken`. This can be used to ensure that the financeAdmin does not accidentally finalize more withdrawals than the available funds in the contract. Note that `_maxAmountOfToken` is the amount without fees subtracted that is finalized.
5. `extract()`: The financeAdmin can call this to extract any amount of USDT from the contract. It is intended to be used when a larger amount of USDT than necessary to cover withdrawals has been transferred to this contract.
6. `getWithdrawalRequestIds()`: returns all unclaimed requestIDs for an address.
7. `lockedToken()`: The `lockedToken` variable tracks the amount of USDT (with fees already subtracted) that are owed to users in finalized withdrawalRequests. If the contract currently holds a balance of USDT smaller than `lockedToken`, it means that not all finalized withdrawalRequests can be claimed immediately.

Additionally, there are setters for parameters and roles, that can be called only by the admin role.

Note that there is no explicit way for admins to withdraw the withdrawal fees accrued in the contract. Instead, the admins can implicitly claim the fees by only depositing an amount of USDT for withdrawals that already has the fees subtracted, or by reclaiming fee amounts using `extract()`.

2.2.4 WstUSDTG2

WstUSDT is a wrapper for stUSDT. WstUSDT balances are non-rebasing, which can be useful for compatibility with other smart contract systems. WstUSDT balances are equivalent to shares of stUSDT. WstUSDT has only the proxy admin role.

WstUSDT implements all functions defined in TRC20/ERC20. Additionally, it implements the following most important functions:

1. `wrap()`: Allows a user to exchange an amount of stUSDT for WstUSDT. The amount received by the user will be exactly the number of stUSDT shares that represented their stUSDT amount.
2. `unwrap()`: Allows a user to exchange WstUSDT back to stUSDT.

The WstUSDTG2 contract is equivalent to WstUSDTG1.

2.2.5 AdminProxy

All contracts listed above are deployed as upgradeable proxies. This means that the implementation contracts can be changed at any time by the `admin` role, which could change any of the behavior.

The proxies all use the same inheritance pattern. We will illustrate it using the StUSDTProxy:

- Proxy: stUSDTProxy is AdminProxy, AdminProxy is AdminStorage
- Implementation: StUSDTG1 is StUSDTStorage, StUSDTStorage is AdminStorage

As both the proxy and implementation have AdminStorage as their most derived contract, they both have the same storage layout for storage slots 0 to 3:

- Slot 0: `address public admin`
- Slot 1: `address public pendingAdmin`
- Slot 2: `address public implementation`

- Slot 3: `address public pendingImplementation`

In case the implementation contracts are ever upgraded, extreme care must be taken not to accidentally change the storage layout of the implementation (for example by changing the inheritance order). If the storage layout of a new implementation does not match that of the proxy, this could lead to a critical vulnerability. See also Proxy Upgrades Must Be Well-tested.

Whenever the `admin` role is referenced in one of the implementation contracts, this is the same as the proxy `admin`.

AdminProxy implements the following most important functions:

1. `_setPendingImplementation()`: The admin can call this to set a new implementation that they want to upgrade to.
2. `_acceptImplementation()`: The pending implementation can call this to become the new implementation contract and replace the previous one.
3. `_setPendingAdmin()`: The admin can call this to set a new pendingAdmin.
4. `_acceptAdmin()`: The pendingAdmin can call this to accept the admin role and replace the previous admin.
5. `fallback()`: Any calls to functions that do not match one of the above (or the public getters), will call the fallback function. The fallback function will `delegatecall` to the implementation contract and forward any calldata to the implementation. It will return the return data given by the implementation.

2.2.6 BlackListManager

The BlackListManager maintains the blacklist used by multiple contracts in the system. It has 2 roles: The admin role and the operator role. The admin role can add operators, and both roles can add addresses to the blacklist. Only the admin can remove addresses from the blacklist.

BlackListManager implements the following most important functions:

1. `addBlackList()`: The admin or an operator can call this to add addresses to the blacklist.
2. `removeBlackList()`: The admin can call this to remove addresses from the blacklist.
3. `isBlackListed()`: Callable by anyone. Returns `true` if the given address is on the blacklist, and `false` otherwise.

Additionally, there are setters for the operator and admin roles, that can be called only by the admin role.

2.2.7 MinterPauseController

PauseController inherits from the OpenZeppelin AccessControl library. It is used to define roles, some of which have the right to pause minting on a specific minter. The minter must configure the PauseController as its pausedAdmin. There are two roles defined in the constructor: `DEFAULT_ADMIN_ROLE`, `OPERATOR_ROLE`, and `MONITOR_ROLE`. The `DEFAULT_ADMIN_ROLE` can add/remove addresses to the `OPERATOR_ROLE`, while the `OPERATOR_ROLE` can add/remove addresses to the `MONITOR_ROLE`.

In addition to the functions defined by OpenZeppelin's AccessControl, it implements the following functions:

1. `addMonitors()`: Callable by the `OPERATOR_ROLE`. Batches the addition of multiple addresses to the `MONITOR_ROLE`.
2. `start()`: Callable by the `DEFAULT_ADMIN_ROLE` or `OPERATOR_ROLE`. Unpauses the minter if it is paused.
3. `pause()`: Callable by the `MONITOR_ROLE`. Pauses the minter if it is not already paused.

4. `setMinter()`: Callable by the `DEFAULT_ADMIN_ROLE` or `OPERATOR_ROLE`. Provides the minter which should be paused. Can only be called once, which means it is not possible to change the minter bound to a pause controller.

The following function has been added to AccessControl:

1. `replaceRole()`: Allows an address that has a role to revoke the role from itself and transfer that role to another address.

StUSDTPauseController is equivalent to MinterPauseController, except that it used to pause all minting of stUSDT instead of just a specific minter. It does not implement the `setMinter()` function.

2.2.8 SwapRouterForStUSDT

SwapRouterForStUSDT allows a user to turn USDT into wstUSDT in one call.

It implements the following functions:

1. `usdtToWstUSDT()`: Takes USDT from the user, converts them to stUSDT, and wraps them to wstUSDT. Sends the resulting wstUSDT to the user.

2.2.9 SwapRouterForJustlend

SwapRouterForJustlend provides helper functions to interact with stUSDT lent or borrowed on JustLend.

It implements the following functions:

1. `usdtToJwstUSDT`: Takes USDT from the user, converts it to wstUSDT, and deposits it on JustLend. Sends the resulting jwstUSDT back to the user.
2. `stUSDTToJwstUSDT`: Takes stUSDT from the user, converts it to wstUSDT and, deposits it on JustLend. Sends the resulting jwstUSDT back to the user.
3. `repayByUSDT()`: Takes USDT from the user, and repays a debt denominated in wstUSDT on JustLend.
4. `repayByStUSDT()`: Takes stUSDT from the user, and repays a debt denominated in wstUSDT on JustLend.

2.2.10 ValuesAggregator

ValuesAggregator is a helper contract, intended for use with frontends.

It implements the following view functions:

1. `getAmountPerShare()`: Returns the stUSDT value of a given number of stUSDT shares, rounded up.
2. `getAmountLimit()`: Returns the minimum amount of USDT that can be staked at once, the minimum amount of stUSDT that can be withdrawn at once, and the minimum amount of TUSD that can be staked at once.
3. `getBalanceAndApprove()`: For a given user and a list of pairs of tokens and pools, returns a struct with the user's balance of the token and the allowance that the pool address has to spend the token of the user. The token address `0x0` can be given to query the native token (TRX/ETH).
4. `getBalanceAndApprove2()`: Similar to `getBalanceAndApprove()`, with a different return format. The balance and approval amounts are returned separately, without using a struct.
5. `getBalance()`: For a given user and a list of tokens, returns an array with the balances of the user in those tokens.

2.2.11 Ethereum contracts

The contracts under the subdirectory `FOR_ETHEREUM_CONTRACT/` are equivalent to the ones in the root directory. They are intended to be deployed to Ethereum Mainnet instead of Tron. The only differences are the absence of `MinterForTUSD` and `AssetSwapRouter`, special handling for the missing return value of USDT on Ethereum (using TransferHelper), and renaming of variables (e.g. ERC20 instead of TRC20).

2.2.12 Roles and Trust Model

The admin roles described above are trusted and expected to behave in the interest of users. The most powerful admin role is the `admin`, as it has the power to upgrade the implementations of proxy contracts and define the other roles. The `admin` is expected to only upgrade implementations to well-tested, non-malicious contracts.

The other very powerful role in the system is the off-chain custodian. Once the USDT have been extracted from the Minter contract, the custodian has full control over the assets. The custodian is expected to only invest the assets in ways that the users expect. Any net profits (with applicable fees subtracted) made through the off-chain investments are expected to be paid out to users in the form of rebases. The custodian is expected to honor withdrawal requests made in the UnstUSDT contract as soon as possible.

Underlying Tokens are assumed to be dollar-pegged TRC20/ERC20 stablecoins. The underlyings must have a very stable peg, as the system always values them at exactly one dollar when minting. If an underlying loses peg, minting should be paused immediately. Tokens with transfer hooks, such as ERC777, are not supported as underlyings. Tokens with more than 18 decimals, or tokens that do not implement the decimals() function are also not supported.

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
- Trust: Violations to the least privilege principle

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	1

- Rounding Errors in TRC20 Methods Risk Accepted

5.1 Rounding Errors in TRC20 Methods

Correctness Low Version 1 Risk Accepted

CS-STUSDT-001

StUSDT performs accounting using amounts of shares, not balances.

As a result, passing certain amounts to the `transfer()` or `transferFrom()` functions can result in rounding errors. The effective amount transferred (change in balance) can be smaller than the input amount by a few wei.

This has the following consequences:

- A `transferFrom(amount)` call in a contract may leave the contract with less balance than `amount`. This breaks assumptions generally made for TRC20/ERC20 tokens and could lead to reverts in the calling contract.
- A `transfer()` with amount a to an address that already has b tokens may result in the receiver having fewer than a `+` b tokens.
- When `transferFrom()` is called, the allowance will be reduced by an amount that is rounded down. This means that the user's balance can decrease by a few wei more than is reflected in the allowance. This also applies to `burnShares()`.
- A `Transfer()` event may not reflect the actual change in balance resulting from the transfer.

It is even possible for a `Transfer()` event to be emitted that contains more than the entire balance of the user.

Consider the following example where the rate of shares to stUSDT balance is slightly more than 1-to-1:

1. User A has a balance of 1E18 and calls transfer(1E18 + 1).
2. The shares transferred are rounded down to 1E18.
3. The user's balance correctly decreases by 1E18, but an event Transfer(1E18+1) is emitted.

Contracts integrating with stUSDT should be aware of this behavior. Integrating contracts should not rely on receiving the exact amount of tokens that are given as argument to `transferFrom()`.

Risk accepted:

stUSDT is aware of the rounding issue and accepts the risk it poses to integrating contracts. Other widely used rebasing tokens have similar behavior. The risk posed to stUSDT itself is small, as the worst case is rounding on the order of 1E-18 USD.

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	1

- Uninitialized Memory Array Code Corrected

Low-Severity Findings	3

- Extract Is Not Limited by lockedTokens Code Corrected
- Finalize maxAmount Check Includes Fees Code Corrected
- Inapplicable Functions in Interface Code Corrected

Informational Findings	4

- Missing Functions in Interface Code Corrected
- Missing Input Validation Code Corrected
- pauseController Does Not Use AccessControl as Intended Code Corrected
- Unclear Comments Specification Changed

6.1 Uninitialized Memory Array

Correctness Medium Version 1 Code Corrected

CS-STUSDT-002

In `ValuesAggregator.getBalanceAndApprove()`, the return variable `_allowance` is never initialized. Trying to write to an uninitialized memory array causes a revert.

As a result, the `getBalanceAndApprove()` function will always revert.

In contrast, the other return value `info` is initialized correctly.

Code corrected:

The memory array is now correctly initialized.

6.2 Extract Is Not Limited by lockedTokens

Trust Low Version 1 Code Corrected

CS-STUSDT-004

In `UnstUSDTG1.sol`, the `extract()` function allows the admin to withdraw all USDT from the contract. This includes the `lockedTokens` amount, which the contract knows should be reserved for finalized withdrawals.

The `extract()` function could have an additional check that ensures the extract does not reduce the balance of the contract below `lockedTokens`, so that it can serve all finalized withdrawals. This would slightly reduce the trust required in the financeAdmin.

However, the impact of this addition would be small as the system requires full trust in the admins anyway.

Code corrected:

The `extract` function can now only withdraw tokens that are not in `lockedTokens`. This means the financeAdmin cannot withdraw funds that are needed for finalized withdrawals.

6.3 Finalize maxAmount Check Includes Fees

Design Low Version 1 Code Corrected

CS-STUSDT-003

In `_finalize()`, the `maxAmountOfToken` check compares the `finalizedAmount` of all new finalized requests to the max given by the admin.

This is intended to be used so that the admin can ensure the contract has enough funds to cover all new finalized requests. However, the contract only needs enough to cover the requests minus fees.

It may make more sense to compare the `maxAmountOfToken` to the finalized amounts with fees subtracted, instead of the full amount.

Code corrected:

Fees are now subtracted from the `finalizedAmount` before being compared to `maxAmountOfToken`.

6.4 Inapplicable Functions in Interface

Correctness Low Version 1 Code Corrected

CS-STUSDT-005

In `ValuesAggregator.sol`, the `IStUSDT` interface defines the `vote_user_power()` and `vote_user_slopes()` functions which are not present in the current StUSDT contract.

Code corrected:

The functions have been removed from the interface.

6.5 Missing Functions in Interface

Informational Version 1 Code Corrected

CS-STUSDT-007

The `IStUSDT` interface is missing some of the public functions that `StUSDTG1` implements.

Code corrected:

The missing functions have been added to the interface.

6.6 Missing Input Validation

Informational | Version 1 | Code Corrected

CS-STUSDT-008

In `UnstUSDTG1.setFinanceAdmin()`, the input address is allowed to be zero.

Other setters validate that addresses must be non-zero.

Code corrected:

A zero check has been added.

6.7 Unclear Comments

Informational | Version 1 | Specification Changed

CS-STUSDT-011

In `AssetSwapRouter` line 86, a comment ends with "(see" which suggests that the end of the sentence has been cut off.

In the comment in `UnstUSDTG1` at line 372, there is a Chinese comment, whereas the rest of the codebase is in English.

In `FOR_ETHEREUM_CONTRACT/MinterG1.sol` line 92, the comment is not applicable since it refers to the TRON version of USDT rather than the Ethereum version.

Specification changed:

All mentioned comments have been updated.

6.8 pauseController Does Not Use AccessControl as Intended

Informational | Version 1 | Code Corrected

CS-STUSDT-012

In `PauseController.sol`, the `addMonitors` function uses `_setupRole()` of AccessControl to grant the `MONITOR_ROLE` to addresses.

This works, but it does not follow the intended flow of OpenZeppelin's AccessControl. The OpenZeppelin repo contains the following comment:

```
*[WARNING]
* ====
* This function should only be called from the constructor when setting
* up the initial roles for the system.
*
* Using this function in any other way is effectively circumventing the admin
* system imposed by {AccessControl}.
* ====
```

The `_setupRole()` function essentially hardcodes the `OPERATOR_ROLE` as admin of `MONITOR_ROLE`. Admin roles are intended to be flexible in the AccessControl framework.

Code corrected:

`addMonitors` now calls `Accesscontrol.grantRole()` and does not perform its own access control.

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 Possible Gas Optimizations

Informational **Version 1**

CS-STUSDT-006

The following is a list of possible gas optimizations:

1. In `StUSDTG1.sol`, the storage value `shares[owner]` is read twice. It could be cached in memory to save an SLOAD operation.
2. In `UnstUSDTG1.sol`, all fields of the WithdrawalRequest `preRequest` are loaded from storage to memory. However, only the `cumulativeStUSDT` and `cumulativeFee` fields are read. Not loading the unused fields could save an SLOAD operation.
3. In `AdminProxy.sol`, the `acceptImplementation()` function checks that the `pendingImplementation` is `msg.sender` AND that the pendingImplementation is not `0x0`. However, `msg.sender` can never be `0x0`, so the second check is redundant. The same is the case in `_acceptAdmin()` in `AdminProxy.sol` and in the `acceptAdmin()` function of `BlackListManager.sol`.

Code corrected:

The code has been changed to implement some gas optimizations. The AdminProxy contract has not been optimised because it is non-upgradeable.

7.2 Some Decimals Values Unsupported by Minter

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

CS-STUSDT-009

In `MinterG1.sol`, the constructor sets the conversionFactor as follows:

```
uint256 decOfToken = _token.decimals();
conversionFactor = 10 ** (decOfStUSDT - decOfToken)
```

The `decimals()` function is optional in the TRC20/ERC20 standard. Tokens that do not implement the `decimals()` function cannot be used with MinterG1. Additionally, tokens with more decimals than stUSDT (18) will cause an underflow in the constructor and are also not supported.

Acknowledged:

stUSDT acknowledged the issue and stated that they only plan on using tokens as underlying that have up to 18 decimals.

7.3 SwapRouterForJustlend Infinite Allowance Can Be Griefed

Informational Version 1 Acknowledged

CS-STUSDT-010

At construction, SwapRouterForJustLend gives "infinite" allowance (`uint256.max`) to the WstUSDT contract to allow for normal operation. As the contract is used, the allowance will be used and could eventually reach zero.

Under normal usage, this is unlikely to happen in a realistic timeframe. However, a dedicated griefer could cycle the same funds through the contract to deplete the allowance, by repeatedly calling `stUSDTToJwstUSDT` and redeeming jwstUSDT back to stUSDT. The only cost to the griefer would be the gas cost. The griefer could use flash loans to obtain more capital.

If the allowance reaches zero, most functions of the SwapRouter will become unusable.

Acknowledged:

stUSDT acknowledged the issue and stated they think it is unlikely that such an attack would be possible in a realistic timeframe.

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 Admin Must Pause Minting in the Event of Stablecoin Depeg

Note **Version 1**

If a supported stablecoin is trading under one dollar, it becomes instantly profitable for users to mint StUSDT using the stablecoin's minter.

However, the admin may not be able to redeem the stablecoin in question at par. This will cause StUSDT to become undercollateralized, and also lose its dollar peg.

As a result, the admin should monitor supported stablecoins closely and quickly pause the Minter of any input stablecoin that has depegged.

8.2 Modified Compiler Should Be Carefully Tested

Note **Version 1**

The smart contracts that will be deployed to Tron will be compiled with the solidity-Tron compiler, which is a modified version of the Ethereum solidity compiler. As the compiler is modified, it might not have the same security level as the original compiler.

The low-level assembly code that is used, for example in the proxies, makes certain assumptions about how the compiler behaves, such as the location of the free memory pointer.

To ensure that the compiler behaves as assumed, we recommend using a testing environment where the implementation contracts are deployed using proxies, instead of testing the implementation contracts in isolation.

8.3 Proxy Upgrades Must Be Well-Tested

Note **Version 1**

The proxy pattern used in the project contains functions and variables in the proxy contract. This proxy pattern has some caveats that admins should be aware of.

1. Storage layout The storage layout is not hardcoded (as it would be if EIP1967 was used). This means extreme care must be taken whenever a proxy is upgraded. The new implementation contract's storage layout must still match the proxy's storage layout exactly. If the storage layout changes, critical variables such as the `implementation` address or the `admin` may be accidentally overrideable.

2. Function selectors The proxy contract contains functions (the state changing functions, and the getters for the public state variables). This means that any calls matching the function selectors of the proxy will not be forwarded to the implementation, but executed in the proxy itself. This can happen if the implementation has a function with the same name as the proxy's functions. However, this is not the only case where it can happen. The function selectors could also randomly collide (they are only 4 bytes long). This would result in the function on the implementation being impossible to call, until the implementation is upgraded again to fix the collision.

The function selector problem could be avoided by using a "transparent proxy" or "UUPS" pattern. However, it would require a redeployment of the current proxies.

Whenever a proxy upgrade is planned, the admins should ensure proper testing. This should include a forked-mainnet test that executes the deploy script for the new implementations and ensures:

1. The storage layout of the new implementation matches the old implementation (with an optional addition of new slots at the end).
2. The new implementation's function selectors do not collide with those of the proxy contract (except for getters of the same variable, which are expected to collide).

The current implementation contracts have been checked for both properties.

8.4 TRC-10 Might Get Stuck

Note **Version 1**

The stUSDT contracts do not support TRC-10 tokens, the built-in token class of the Tron network.

Note that any TRC-10 tokens sent to a contract that is deployed as a proxy will be stuck, as they will be accepted, but cannot be transferred onwards. However, this would only apply to accidentially sent TRC-10 tokens.

8.5 Underlying Tokens Must Not Have Transfer Hooks

Note **Version 1**

`MinterG1.submit()` (and many other functions) account for the amount of tokens transferred from the user by comparing the balance before and after the `transferFrom()` call.

If a token with transfer hooks (such as ERC-777) was used, a reentrancy attack could mint unbacked stUSDT.

Consider the following example attack, assuming the underlying `token` implements ERC-777:

1. Call `submit()` with `100` tokens.
2. Use the `tokensToSend()` hook in `token.transferFrom()` to call `submit()` with 100 again.
3. The inner call will correctly mint `100` stUSDT and return.
4. In the first call, `newBalance - oldBalance` will be `200`, so `100` unbacked stUSDT will be minted.

The same issue exists in `MinterForTUSD`.

The admin can mitigate this risk by ensuring that any token that is added as underlying for a new Minter does not contain transfer hooks.

8.6 Withdrawal Race Condition

Note **Version 1**

In `UnstUSDTG1.sol`, it is possible that more withdrawals are finalized than the amount of USDT that the contract holds. This is especially likely in the case that `finalizingPaused` is set to `true`.

If this situation happens, there will be a race condition on withdrawals, where those users that call `claimWithdrawals()` first will be able to withdraw, while those that are slower will need to wait until the admins deposit additional USDT to the contract.

If a situation arises where large outflows are expected, the admins should disable `finalizingPaused` and only finalize as many withdrawals as the contract has funds. This will ensure a first-in, first-out order of processing withdrawals.