PUBLIC

Limited Code Review

of the Trinity Smart Contracts

7 June, 2024

Produced for

by

Contents

1 Executive Summary 3
2 Review Overview 5
3 Limitations and use of report 9
4 Terminology 10
5 Findings 11
6 Resolved Findings 13
7 Informational 19
8 Notes 21

1 Executive Summary

Dear Trinity team,

Thank you for trusting us to help Trinity with this security review. Our executive summary provides an overview of subjects covered in our review of the latest reviewed contracts of Trinity according to Scope to support you in forming an opinion on their security risks. The review was executed by two engineers over a period of two weeks.

Limited reviews are best-effort checks, and do not provide assurances comparable to a non-limited code assessment. Note that only the differences between Gravita and Trinity were in scope, assuming Gravita is bug-free.

Trinity is a protocol designed to facilitate borrowing against yield-bearing collateral. Borrowers mint TRI, a dollar-based token that can be used to take leveraged T-Bill positions and capture Trinity protocol fees through staked TRI (sTRI).

The most critical subjects covered are correct accounting, correctness of the liquidation and redemption mechanisms, and correctness of the fees and their distribution. Accounting correctness was improved, as the issue Vessel Fees Are Not Added to Global Debt was fixed. Correctness of the redemption and liquidation mechanism was low, see Redemptions Are Not Possible in Recovery Mode and Liquidations Are Not Disabled. In response to this, there was a major specification change during the review period. Correctness of the fees is improvable, see Borrowing fees are not applied before closing a vessel and Borrowing fees need to be triggered every epoch.

The general subjects covered are testing and documentation. Testing could be improved, as many functional issues were uncovered that could have been found through rigorous testing. Documentation could be improved, as some changes made are not yet documented in detail.

As the goal of this limited review was to provide time-bound security insights on a complex codebase in a limited time, and as a large number of issues were uncovered, we refrain from assigning a specific overall level of security to the codebase.

It is important to note that security reviews 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	2
• Code Corrected	1
• Specification Changed	1
Medium-Severity Findings	10
• Code Corrected	4
• Specification Changed	3
• Risk Accepted	3
Low-Severity Findings	1
• Code Corrected	1

2 Review 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

This review was not conducted as an exhaustive search, but rather as a best-effort sanity check. It was performed on the source code files inside the two Trinity repositories, based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

Trinity is a fork of Gravita, and the review was performed only on the differences between Trinity and Gravita, assuming the Gravita codebase to be secure. The commit of Gravita at which it was forked is `0ff6c043af62b181872f3d5c78593e6965e17ca6`.

Trinity-sc (forked)

V	Date	Commit Hash	Note
1	08 April 2024	71eebd3fe9aeaaaec6ca4a6678e1bd9ca36c8d33	Initial Version
2	12 April 2024		Same code, specification changed
3	13 May 2024	322cba2fc63ae6c77877335fbbb272a91c7e92b5	Version 3
4	30 May 2024	7022d480bd105a69a4e9f0bba95037cb273d46ab	Version 4
5	07 June 2024	a2153e81be66f6feefeba5d8cc309a019e20341f	Final Fixes

Trinity (staking)

V	Date	Commit Hash	Note
1	08 April 2024	384c90ae81f715bd47b051d32a577907504d7f40	Initial Version
2	12 April 2024		Same code, specification changed
3	13 May 2024	6e3d6635cf6770b141f1cc594439c337c5fe9432	Version 3

For the solidity smart contracts, the compiler version `0.8.20` was chosen.

The contracts in scope are:

Core protocol (only differences from Gravita):

- contracts/ActivePool.sol
- contracts/AdminContract.sol
- contracts/BorrowerOperations.sol
- contracts/DebtToken.sol

- contracts/StabilityPool.sol
- contracts/VesselManager.sol
- contracts/VesselManagerOperations.sol
- contracts/Pricing/ERC4626PriceFeed.sol

Staking ERC4626 vault:

- packages/contracts/src/SavingsTRI.sol
- packages/contracts/src/Distributor.sol

2.1.1 Excluded from scope

All contracts that are not mentioned in the scope are automatically considered to be out of scope. In particular, the tfBill fund contracts are out of scope, and the implementation of the user verification mechanism in the staking contract is out of scope.

Any issues that were already present in Gravita when the code was forked are out of scope, as only the differences were considered.

Any economic attacks are out of scope.

2.2 System Overview

This system overview describes the revised specification (Version 2) of the contracts as defined in the Review Overview. The contracts of Version 1 and Version 2 are equivalent, but there were major specification changes.

At the end of this report section, we have added subsections for each of the changes according to the versions, including the initial spec which was changed during the review process.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Trinity is a protocol designed to facilitate borrowing against yield-bearing collateral. Trinity is a fork of Gravita, which is a fork of Liquity that allows the use of ERC20 collateral, not just ETH. It will initially be deployed to the Base Blockchain.

2.2.1 Major modifications compared to Gravita

Trinity is designed to be used with low-volatility collateral, such as tokenized US T-Bills. Trinity operates under the assumption that the value of the collateral will not drop significantly. If it does, the TRI token will depeg and trade below 1 USD.

The major modifications compared to Gravita are the following:

1. The liquidation, stability pool, and redemption mechanisms have been modified to only be accessible by whitelisted actors.
2. Redemptions are expected to be disabled by default and will be enabled by Governance in case the TRI token trades below 1 USD even though the system is overcollateralized.
3. Redemption fees can be configured to have a constant value. This disables the fee increase when there are many redemptions in a short period, which is present in Gravita.
4. The `debtTokenGasCompensation` parameter will be set to 0, as it is assumed that liquidations will only very rarely happen.

5. The CCR parameter will be set to 0, meaning the system will never enter recovery mode. It will always operate in normal mode.
6. The one-time minting fee has been changed to a recurring weekly fee, paid upfront for the coming week.
7. Fees are no longer refunded if a user repays their position early.
8. All borrow fees generated by Trinity are sent to the Distributor contract, which distributes them to the SavingsTRI vault.
9. The collateral value is determined using the `convertToAssets` function of the collateral token, not by directly calling a price oracle such as Chainlink.
10. The governance token (GRVT in Gravita) has been removed and there are no governance tokens issued to StabilityPool depositors.

It is also assumed that the borrow fee will be set to a value that is lower than the yield of the T-Bills underlying the collateral. This means the collateralization ratio of all vessels should improve over time (assuming collateral value does not fall).

2.2.2 SavingsTRI

SavingsTRI is an ERC4626 Vault. Users can deposit their TRI to receive sTRI. Over time, borrow fees paid to the Trinity protocol will be paid out to sTRI holders, proportional to their share of the total sTRI supply.

sTRI can only be minted, redeemed and transferred to or from addresses that pass the `isAllowed()` check of the UserVerifier contract. This check is meant to verify that the address belongs to a non-US user. The UserVerifier is treated as a black box for this review, as it is not currently publicly available.

SavingsTRI has an admin role that can set the `epochLength`, the `MaxApy`, and the `receiver` address, which will receive the distributions. It can also upgrade the implementation of the contract.

2.2.3 Distributor

The Distributor contract is responsible for distributing fees generated by the Trinity protocol to the SavingsTRI vault. At the beginning of each epoch, the `distributionPerSecond` is determined, which is the rate at which the vault receives rewards during that epoch. The rate is chosen such that by the end of the epoch, the balance of the Distributor would reach zero if it did not receive any additional tokens later. There is also a `MaxApyDistribution` parameter that limits the maximum APY that can be distributed to the vault. If the APY would exceed this value, the excess will be distributed in a later epoch instead.

The Distributor has an admin role that can set the `epochLength`, the `MaxApy`, and the `receiver` address, which will receive the distributions (intended to be the SavingsTRI vault). It can also upgrade the implementation of the contract.

2.2.4 Trust Model

The owner of the Trinity core contracts (forked from Gravita) is fully trusted. It can upgrade the implementations. In the worst case, this would allow stealing all collateral deposited in the system and minting an unlimited number of TRI tokens.

The owner of the DebtToken contract is fully trusted. It can add addresses to a whitelist that are allowed to mint TRI without any collateral. In the worst case, this could lead to minting an unlimited number of TRI tokens.

The admin of the SavingsTRI contract is fully trusted. It can pause the contract, making it impossible for users to withdraw. Additionally, it can upgrade the implementation. In the worst case, this would allow stealing all TRI deposited in the Vault.

The admin of the Distributor contract is partially trusted. It can upgrade the implementation. In the worst case, this would allow stealing all pending rewards (but not Vault deposits) and making withdrawals from the sTRI contract impossible, due to reverts in `claim()`. If this happens, withdrawals could be reenabled by the sTRI admin, by upgrading the sTRI implementation.

The `convertToAssets` function of the collateral tokens, which is used as price oracle, is fully trusted. If this function returns a price that is too high, it would be possible to mint undercollateralized TRI. In the worst case, an unlimited number of TRI tokens with very little collateral could be minted. Additionally, it is assumed that the underlying of the collateral token is non-transferable and does not trade on secondary markets.

The collateral tokens are assumed not to significantly fall in value. If they do, the TRI token will depeg and trade below 1 USD.

Collateral tokens should not have any non-standard behavior, such as rebasing or fees-on-transfer.

The whitelisted addresses (liquidators and redeemers) are assumed to act in the best interest of the protocol.

2.2.5 Initial specification

At the beginning of the review, the initial specification (Version 1) was different from Version 2 in the following ways:

1. Liquidations were meant to be disabled.
2. Redemptions were meant to be possible in both normal and recovery mode.
3. The `CCR` parameter was meant to be set to a value above 1, meaning the system was able to enter recovery mode.

2.2.6 Changes in Version 3

In Version 3, the gas compensation logic was completely removed from the code, as it was not intended to ever be used.

3 Limitations and use of report

Security reviews cannot uncover all existing vulnerabilities; even a review in which no vulnerabilities are found is not a guarantee of a secure system. However, code reviews 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 review aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed a review in order to discover as many vulnerabilities as possible.

The focus of our review was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These reviews 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 review, 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	3

- Borrowing Fees Are Not Applied Before Closing a Vessel Risk Accepted
- Borrowing Fees Need to Be Triggered Every Epoch Risk Accepted
- Distributor Rewards Are Not Distributed Evenly Risk Accepted

Low-Severity Findings	0

5.1 Borrowing Fees Are Not Applied Before Closing a Vessel

Correctness Medium Version 1 Risk Accepted

CS-TRI-003

To close a vessel, a user can call the `BorrowerOperations.closeVessel` function. However, this function does not call `_collectVesselFee()`. As a consequence, it is not guaranteed that the weekly fee will be applied before closing the vessel. This also implies that the debt to repay in order to close the vessel will be lower than it should be.

Note that this issue is aggravated since weekly fees are not applied if not manually triggered. It is possible for a user to open a vessel and pay a one-time fee on the amount borrowed. Then, after an arbitrary amount of time, the user will be able to close their vessel and repay the loan without having to pay any interest proportional to the loan duration, unless someone else triggers `_collectVesselFee` on the vessel in the meantime.

Risk accepted:

Trinity responded:

```
This functionality is by design. As all of the borrowing fees are charged up front for a week,
we do not want to charge any additional fees for closing the vessel. After the epoch ends and
before we trigger the fees manually, there might occur a small time frame where it would be
possible for a user to avoid paying for the next week, but the loss is negligible.
```

5.2 Borrowing Fees Need to Be Triggered Every Epoch

Design | Medium | Version 1 | Risk Accepted

CS-TRI-002

Trinity adds a weekly borrow fee to active vessels. To determine whether a vessel has already paid its fee for the week, its `lastFeeCollectionEpoch` is stored in a mapping. When `_collectVesselFee()` is called, the fee is charged if the vessel has not paid its fee for the current epoch.

However, if the fee was last collected more than one epoch ago, the fee is only charged once. In this case, the vessel would pay fewer fees than it should.

`_collectVesselFee()` should charge the user for the number of epochs since the last fee collection.

Risk accepted:

Trinity responded:

```
As the fees need to be triggered per vessel, there is no easy way to account for all of them at once.
As the protocol maintainers, we commit to triggering the fees manually (preferably by using bots).
```

5.3 Distributor Rewards Are Not Distributed Evenly

Design | Medium | Version 1 | Risk Accepted

CS-TRI-008

Assuming that the distributor contract receives a constant amount of TRI per time unit, the rewards in `Distributor` are supposed to be distributed evenly to the staking vault during the course of epochs. At the beginning of each epoch, the current balance of the contract and the epoch length are used to compute how much should be distributed per second during that epoch.

However, note that a new epoch (n+1) does not start at the end of the previous one (n): it starts when the function `claim` is called for the first time after `endEpochTimestamp` of epoch n.

This means that the amount to be distributed in epoch n+1 includes all the rewards skipped between `endEpochTimestamp` of epoch n and the beginning of epoch n+1, as this time does not belong to any epoch. These rewards will accumulate in the contract and get distributed during epoch n+1, making is so that that the amount that gets distributed during each epoch is not even. It varies based on when the epoch is started. Some time periods will not belong to any epoch.

Risk accepted:

Client responded:

```
This functionality is intended. The distribution is to be updated as often as possible
(deposits, withdrawals or manually using bots). As a safety measure, we implemented
the APY cap feature not to distribute significant funds too quickly.
```

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	2

- Redemptions Are Not Possible in Recovery Mode Specification Changed
- Vessel Fees Are Not Added to Global Debt Code Corrected

Medium-Severity Findings	7

- Borrowing Fees Are Paid in Recovery Mode Specification Changed
- Distributor Can Run Out of Tokens Code Corrected
- Gas Compensation Cannot Be Disabled Code Corrected
- Liquidations Are Not Disabled Specification Changed
- Redemption Fees Are Locked in Distributor Code Corrected
- SavingsTRI Does Not Correctly Implement ERC4626 Code Corrected
- Some Vessels Cannot Be Redeemed Specification Changed

Low-Severity Findings	1

- Some Parameters Cannot Be Set to the Intended Values Code Corrected

Informational Findings	1

- Outdated Documentation Specification Changed

6.1 Redemptions Are Not Possible in Recovery Mode

Correctness High Version 1 Specification Changed

CS-TRI-018

According to Version 1 of the specification, redemptions were meant to be possible both in recovery mode and in normal mode.

However, in recovery mode redemptions were disabled.

This would pose a problem for the peg stability of TRI, as the redemption mechanism supports price recovery of TRI in the case where it is trading below 1 USD.

Specification changed:

In Version 2 of the specification, the CCR parameter is meant to be set to zero. As a result, the system can never enter recovery mode. This resolves the issue, as it only applies to recovery mode.

6.2 Vessel Fees Are Not Added to Global Debt

Correctness | High | Version 1 | Code Corrected

CS-TRI-001

In `BorrowOperations.collectVesselFee()`, the borrower fee is added to the user's debt, but not to the global debt accounting of the active pool. When user debt gets repaid, the function `ActivePool.decreaseDebt()` removes the user debt from the active pool debt.

Since the user debt includes fees, removing all user debt from the pool would underflow the debt of the active pool and the transaction would revert. This would make it impossible to close some vessels and funds would be stuck in the contract, unless the admin upgrades the implementation.

The borrow fee should be correctly added to the active pool debt to make the accounting consistent.

Code corrected:

In Version 3 the code was updated such that when the external function `collectVesselFee` is called, the borrowing fee is accounted globally:

```
function collectVesselFee(address _asset, address _borrower) external {
            uint256 collectedFee = _collectVesselFee(_asset, _borrower);
            IActivePool(activePool).increaseDebt(_asset, collectedFee);
    }
```

However, the correct global accounting of the fee only happened when calling the **external** function `collectVesselFee`. When the **internal** function `_collectVesselFee` was called, the borrowing fee was not accounted globally. In addition, any call to `_collectVesselFee` would set `lastFeeCollectionEpoch` to the current epoch, disabling the external `collectVesselFee` function for that epoch. Therefore, if during a given epoch `_collectVesselFee` is called, then `collectVesselFee` would not yield any effect, and would not update the global debt.

In Version 4 the code was updated such that the borrowing fee is correctly accounted globally when calling the internal function `_collectVesselFee`.

6.3 Borrowing Fees Are Paid in Recovery Mode

Correctness | Medium | Version 1 | Specification Changed

CS-TRI-016

In Version 1 of the specification, the system was allowed to enter recovery mode. In that case, the borrowing fees were supposed to be disabled when opening a new vessel. However, it was still possible for anyone to trigger `collectVesselFee` on the new vessel within the same epoch, de facto applying the borrowing fee that was skipped when opening the vessel.

Specification changed:

In Version 2 of the specification, the CCR is set to zero, which prevents the system from entering recovery mode. As a consequence, the borrowing fees are always applied normally.

6.4 Distributor Can Run Out of Tokens

Correctness | Medium | Version 1 | Code Corrected

The Distributor should set the `distributionPerSecond` to a value such that it has enough balance to distribute this amount for a full epoch.

The calculation is done as follows:

If `amountToDistribute` is more than zero, the Distributor will transfer some TRI to the receiver. However, these TRI are not subtracted from the Distributor's balance in the `distributionPerSecond` calculation. As a result, the Distributor will not have enough balance to distribute the full amount for the next epoch, unless it receives more tokens in the meantime.

Consider the following example:

- epochLength: 100 seconds
- Distributor balance: 101 tokens
- distributionPerSecond: 1 token
- lastClaimTimeStamp: 300 seconds
- epochEndTimeStamp: 400 seconds

At 401 seconds, we call `claim()`. The new values will be:

- Distributor balance: 1 token
- distributionPerSecond: 1.01 tokens
- lastClaimTimeStamp: 401 seconds

The distributionPerSecond is 1.01, but it should actually be 0.01 tokens per second. So if we `claim()` at 402 seconds, the contract will try to transfer 1.01 tokens even though it only has a balance of 1 token.

If the Distributor runs out of tokens, the `claim` function will revert. As `claim()` is called on every deposit or withdrawal of the sTRI vault, this will make it impossible for users to withdraw from sTRI. This can be resolved by sending TRI token to the Distributor.

Code corrected:

Trinity has corrected the code in the following way:

```
if (block.timestamp > epochEndTimestamp) {
    epochEndTimestamp = block.timestamp + epochLength;
    uint256 balanceLeftAfterDistribution = TRI.balanceOf(address(this)) - amountToDistribute;
    distributionPerSecond = balanceLeftAfterDistribution / epochLength;
}
TRI.transfer(receiver, amountToDistribute);
```

The `distributionPerSecond` is now computed using the current balance of the distributor minus the amount to distribute from the previous epoch.

6.5 Gas Compensation Cannot Be Disabled

Design | Medium | Version 1 | Code Corrected

CS-TRI-005

According to the specification, the gas compensation in Trinity is supposed to be disabled by setting the `AdminContract.getPercentDivisor` parameter to zero.

However, in TrinityBase, the `_getCollGasCompensation` function divides by the getPercentDivisor. This means that if the `percentDivisor` is set to zero, a division by zero will cause a revert.

Additionally, the AdminContract enforces that the `percentDivisor` must be set to a value between 2 and `200`, so it is not possible to set a value of zero.

Code corrected:

The gas compensation logic has been completely removed from the contracts.

However, there is still the `debtTokenGasCompensation` parameter in the `CollateralParams` of `AdminContract`, which is now unused.

6.6 Liquidations Are Not Disabled

Correctness | Medium | Version 1 | Specification Changed

CS-TRI-019

In Version 1 of the specification, it was stated that liquidations should be disabled.

However, the code was not modified compared to Gravita, and liquidations were still possible.

Specification changed:

In Version 2 of the specification, it was decided that liquidations should be possible by whitelisted addresses.

6.7 Redemption Fees Are Locked in Distributor

Correctness | Medium | Version 1 | Code Corrected

CS-TRI-006

When a redemption occurs, any redemption fees (charged in collateral asset) are sent to `AddressesConfigurable.treasuryAddress`. The borrowing fees are also sent to `treasuryAddress`, assumed to be the address of the Distributor contract. Therefore, redemption fees are sent to the Distributor contract. However, this contract does not offer any way of retrieving assets other than TRI, and as a consequence, the redemption fees will be locked in Distributor. The locked fees could be retrieved by the owner of the Distributor, by upgrading the contract.

Code corrected:

A new `distributorAddress` has been added to the system. Now, the `distributorAddress` receives the borrowing fees, and the separate `treasuryAddress` receives the redemption fees. The `treasuryAddress` will be configured to be able to handle all collateral assets.

6.8 SavingsTRI Does Not Correctly Implement ERC4626

Correctness | Medium | Version 1 | Code Corrected

CS-TRI-007

SavingsTRI does not correctly implement ERC4626 due to the following discrepancies from the specification:

The specification for `maxDeposit()` contains the following:

```
MUST factor in both global and user-specific limits, like if deposits
are entirely disabled (even temporarily) it MUST return 0.
```

In SavingsTRI, the `maxDeposit` function does not factor in that a user may be unable to deposit. This could be the case if the user has not passed the `verifyUser()` check, or if the contract is paused. In these cases `maxDeposit()` should return 0, but it does not. The same issue is also present in `maxMint()`, `maxWithdraw()` and `maxRedeem()`.

Code corrected:

Trinity has corrected the code by overriding the ERC4626 view functions mentioned above and incorporating the following check:

```
if (paused() || !isUserAllowed(receiver)) return 0;
```

6.9 Some Vessels Cannot Be Redeemed

Design | Medium | Version 1 | Specification Changed

CS-TRI-017

In Version 1 of the specification, the system was supposed to only have redemptions and no liquidations. It was intended that a vessel could be redeemed if its individual collateral ratio (`ICR`) was too low.

However, the code did not allow this. The `redeemCollateral` function will start redemptions with the first vessel that has an `ICR` that is greater than the `MCR` (minimum collateral ratio).

```
currentBorrower = ISortedVessels(sortedVessels).getLast(_asset);
    // Find the first vessel with ICR >= MCR //@audit we also want to redeem vessels with ICR < MCR
    while (
            currentBorrower != address(0) &&
            IVesselManager(vesselManager).getCurrentICR(_asset, currentBorrower, totals.price) <
            IAdminContract(adminContract).getMcr(_asset)
    ) {
            currentBorrower = ISortedVessels(sortedVessels).getPrev(_asset, currentBorrower);
    }
```

Vessels with `ICR < MCR` can only be liquidated, not redeemed. As Version 1 intended for liquidations to be disabled, these vessels could only be closed by their owners and could not be redeemed or liquidated.

Specification changed:

In Version 2 of the specification, it was decided that liquidations should be possible. This resolves the issue, as now vessels with `ICR < MCR` can be liquidated.

6.10 Some Parameters Cannot Be Set to the Intended Values

Correctness | Low | Version 1 | Code Corrected

CS-TRI-010

The specification states that "There are no redemption fees in Trinity". The redemption fees are composed of 2 parts: the redemption fee floor and the base rate. The redemption fee floor can be disabled. However, the `setRedemptionFeeFloor` function in the AdminContract requires that the value is set to at least `0.01%`, so it cannot be zero.

In Version 2, the `CCR` parameter is intended to be set to zero. However, because of the following check:

```
safeCheck("CCR", _collateral, newCCR, 1 ether, 10 ether)
```

In AdminContract, the value of the `CCR` is restricted between 1e18 and 10*1e18, which prevents it from being set to zero.

Code corrected:

Trinity has corrected the code: now the function `setRedemptionFeeFloor` can set the redemption fee floor to zero and the function `setCCR` can set the CCR to zero.

6.11 Outdated Documentation

Informational Version 1 Specification Changed

CS-TRI-012

In StabilityPool, the code comments contain a section called "Trinity issuance to stability pool depositors". However, the token issuance to the stability pool that was present in Gravita has been removed in Trinity.

In BorrowerOperations, there is a comment that says "the borrowing fee partial refund is not burned here, as it has already been burned by the FeeCollector". However, the FeeCollector does not exist in Trinity.

The README of the trinity-sc_code repo mentions the FeeCollector contract, even though it was removed in Trinity.

Specification changed:

The incorrect comments have 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 Collateral Decimals Value Could Return Incorrect Value

Informational | Version 1 | Acknowledged

CS-TRI-009

In `AdminContract.addNewCollateral`, the `_decimals` value passed as input is required to be 18. This value will then be stored in the collateral parameters and returned by the getter `getDecimals`. However, collateral tokens with fewer than 18 decimals should also be supported.

This is currently only possible by incorrectly passing 18 as decimals when adding the collateral, which means that the getter would return an incorrect value.

Acknowledged:

Trinity responded:

```
The decimal enforcement has been implemented in the forked protocol by the Gravita team.
While it has to always be strictly set to 18, all of the necessary decimals conversions are
handled by the contracts and the protocol allows accepting tokens with any number of decimal places.
```

7.2 Users That Lose Verification Still Receive Rewards

Informational | Version 1 | Acknowledged

CS-TRI-013

The `SavingsTRI` vault only allows verified users to transfer their sTRI tokens. Consequently, unverified users cannot have sTRI minted or burned, and cannot be the source or the recipient of a transfer.

Note that when a user who was once verified is removed from the verified list, they keep their tokens. The user will keep having a share of the vault and receive reward distributions from borrow fees, but not be able to withdraw.

Acknowledged:

Trinity responded:

```
We find it as a very unlikely scenario for a user to be removed from the whitelist while they
are providing to the Savings TRI vault, but if it ever occurs, we would resolve the issue manually.
```

7.3 openVessel assetAmount Is Unintuitive

Informational | Version 1 | Acknowledged

CS-TRI-015

In BorrowerOperations, the `openVessel` function takes an argument `assetAmount`. The `assetAmount` needs to be given as the amount of assets to deposit, scaled to 18 decimals. For example, if the collateral had 6 decimals and the user wanted to deposit one token, the user would need to input `1E18`, not `1E6` as they would expect.

The documentation could be updated to clarify this behavior.

Acknowledged:

Trinity responded:

```
This issue is caused by the original Gravita design with 18 decimals enforcements.
We handle this for our users using the trinity UI. We assume that a user interacting
with the contract directly is doing so at their own risk.
```

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 Compounding Borrowing Fees

Note Version 1

The borrowing fees are added to the user's debt every week and therefore compound over time. For example, if the weekly borrow rate is set to 0.077 (4 % / 52), and it is charged separately every week, then the debt increases by 4.07% in a year:

$$4.07\% = (1 + 4\%/52)^{52} - 1$$

Governance should take this compounding effect into account when setting the borrowing fees to ensure that the debt does not increase by more than expected.

8.2 Volatile Collateral Could Lead to Depeg

Note Version 1

Trinity is designed to only be used with low-volatility collateral. As the recovery mode is disabled, the system does not incentivize using a collateralization ratio higher than the minimum. If any of the collateral tokens go down in value by more than the `MCR` (Minimum Collateral Ratio), the system will be undercollateralized and the TRI token should trade at a price lower than 1 USD.

E.g. if the `MCR` is 105% and the collateral value drops by 10%, the TRI token should trade around 0.95 USD.

TRI holders should be aware of this risk and ensure that they understand the scenarios in which the value of the enabled collaterals could drop significantly.