

















Ethereum Reserve Dollar (ERD)

Executive Summary

This audit report was prepared by Quantstamp, the leader in blockchain security.

Type	Stablecoin
Timeline	2023-08-14 through 2023-10-05
Language	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	Docs 
Source Code	<ul style="list-style-type: none">Ethereum-ERD/dev-upgradeable #8023739 
Auditors	<ul style="list-style-type: none">Rabib Islam Auditing EngineerIbrahim Abouzied Auditing EngineerHytham Farah Auditing Engineer

Documentation quality	High 
Test quality	High 
Total Findings	33  Fixed: 9 Acknowledged: 21 Mitigated: 3
High severity findings 	7  Fixed: 2 Acknowledged: 3 Mitigated: 2
Medium severity findings 	11  Fixed: 2 Acknowledged: 9
Low severity findings 	8  Fixed: 3 Acknowledged: 5
Undetermined severity findings 	3  Acknowledged: 3
Informational findings 	4  Fixed: 2 Acknowledged: 1 Mitigated: 1

Summary of Findings

Initial report: Quantstamp performed an audit for the contracts behind Ethereum Reserve Dollar (ERD) based on the code present in the linked repository.

The contracts within this repository form the basis for a protocol that issues a stablecoin (USD-pegged asset) called USDE. Users issue USDE by opening and adjusting troves, where they must deposit a certain quantity of collateral in ETH and/or liquid staking derivatives (LSDs). Holders of USDE may choose to redeem the stablecoin for the backing collateral, paying a redemption fee in the process. There is also a liquidation mechanism present which aims to ensure that there is always enough collateral backing the USDE. A trove sorting mechanism has been implemented in an attempt to increase the efficiency of liquidations and redemptions.

Notably, ERD is a fork of Liquity, and has added a number of distinct features. The most noticeable new features include compounding variable interest rate on loans and multiple collateral type support.

We were able to identify numerous areas of focus, including:

1. Correct variable interest rate accrual;

2. Verification of trove sorting mechanism and its impact on redemptions and liquidations;
 3. Tokenization of collateral and the impacts of transferring the token;
 4. Economic considerations;
- and many more.

During the course of the audit, we were able to identify six high severity issues. Among them, perhaps the most concerning from the perspective of likelihood and impact is [ERD-6](#), wherein MEV is generated by liquidations, leading to a flash loan attack. [ERD-5](#) is also very concerning as borrowers will have to essentially forego stETH rebasing rewards in order to participate in ERD.

Writing further tests is also warranted—the test suite should especially contain more tests concerning the variable interest rate accrual mechanism. Integration testing is also well-warranted for a protocol of this size.

We recommend that the client address and/or consider all the findings included in this report.

Update 1: All the issues have been addressed. However, we disagree with the client on a number of issues. Most notably, these are [ERD-4](#) and [ERD-7](#). We are also concerned with the testing situation, as most tests appear not to be working.

Update 2: The available tests are no longer failing.

ID	DESCRIPTION	SEVERITY	STATUS
ERD-1	EToken Transferability Leads to Potential for Collateralization Inaccuracy	• High ⓘ	Fixed
ERD-2	Changing USDE Gas Compensation Causes Discrepancy Between Trove ICRs and TCR	• High ⓘ	Fixed
ERD-3	First Collateral Never Has Its Price Updated	• High ⓘ	Acknowledged
ERD-4	Liquidations Are Not Proportionally Funded by All Troves	• High ⓘ	Mitigated
ERD-5	Insufficient Means for Handling stETH Rebasing	• High ⓘ	Acknowledged
ERD-6	Flashloans Can Be Used to Siphon Liquidation Rewards Through Active Trove Liquidation Mechanism	• High ⓘ	Acknowledged
ERD-7	Oracle Manipulation with Flashloans Can Exploit Funds via Liquidation	• High ⓘ	Mitigated
ERD-8	EToken Transferability May Lead to Undercollateralized Trove	• Medium ⓘ	Fixed
ERD-9	Redemptions and Liquidations Affect Troves in Incorrect Order	• Medium ⓘ	Acknowledged
ERD-10	Maximum Borrow Rate Not Enforced	• Medium ⓘ	Fixed
ERD-11	Upgrades Can Enable Unexpected Loss of Funds	• Medium ⓘ	Acknowledged
ERD-12	Privileged Roles and Ownership	• Medium ⓘ	Acknowledged
ERD-13	Users Have to Pay Interest on the Borrowing	• Medium ⓘ	Acknowledged

ID	DESCRIPTION	SEVERITY	STATUS
	Fee		
ERD-14	USDE Debt May Be Misaccounted For	<ul style="list-style-type: none"> Medium ⓘ 	Acknowledged
ERD-15	Removing Collateral Types Prevents Collateral Withdrawal	<ul style="list-style-type: none"> Medium ⓘ 	Acknowledged
ERD-16	Removing Collateral Types Adversely Affects Collateralization Ratios	<ul style="list-style-type: none"> Medium ⓘ 	Acknowledged
ERD-17	Pending Debt Rewards Do Not Accrue Interest, Disincentivizing Users From Interacting with Protocol	<ul style="list-style-type: none"> Medium ⓘ 	Acknowledged
ERD-18	Unchecked Return Value	<ul style="list-style-type: none"> Medium ⓘ 	Acknowledged
ERD-19	Approximate Hints are Generated Sub-Optimally	<ul style="list-style-type: none"> Low ⓘ 	Acknowledged
ERD-20	Missing Input Validation	<ul style="list-style-type: none"> Low ⓘ 	Fixed
ERD-21	Implementation Contracts Can Be Initialized	<ul style="list-style-type: none"> Low ⓘ 	Fixed
ERD-22	Price Feed May Unnecessarily Resort to Stale Data	<ul style="list-style-type: none"> Low ⓘ 	Acknowledged
ERD-23	When Oracles Are Down, Stale Prices Are Used	<ul style="list-style-type: none"> Low ⓘ 	Acknowledged
ERD-24	Contracts Lack Consolidated Initialization	<ul style="list-style-type: none"> Low ⓘ 	Acknowledged
ERD-25	<code>MIN_NET_DEBT</code> Can Be Set Lower than <code>USDE_GAS_COMPENSATION</code>	<ul style="list-style-type: none"> Low ⓘ 	Fixed
ERD-26	Loss of Precision Due to Division before Multiplication	<ul style="list-style-type: none"> Low ⓘ 	Acknowledged
ERD-27	<code>EToken.transferFrom()</code> May Transfer an Incorrect Amount of Tokens	<ul style="list-style-type: none"> Informational ⓘ 	Fixed
ERD-28	Ownership Can Be Renounced	<ul style="list-style-type: none"> Informational ⓘ 	Mitigated
ERD-29	Missing Initializers	<ul style="list-style-type: none"> Informational ⓘ 	Fixed
ERD-30	Redundant Code	<ul style="list-style-type: none"> Informational ⓘ 	Acknowledged
ERD-31	Interest Rate Increases as Total Collateralization Rate Rises	<ul style="list-style-type: none"> Undetermined ⓘ 	Acknowledged
ERD-32	Frontend Functionality May Be Superfluous for ERD	<ul style="list-style-type: none"> Undetermined ⓘ 	Acknowledged

ID	DESCRIPTION	SEVERITY	STATUS
ERD-33	Redemption of Troves with Lower ICR Creates Incentive for Increasing TCR	• Undetermined ⓘ	Acknowledged

Assessment Breakdown

Quantstamp's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

Disclaimer

Only features that are contained within the repositories at the commit hashes specified on the front page of the report are within the scope of the audit and fix review. All features added in future revisions of the code are excluded from consideration in this report.

Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

1. Code review that includes the following
 1. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
 2. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
 3. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.
2. Testing and automated analysis that includes the following:
 1. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
 2. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarity, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Scope

The scope includes all the contracts in `contracts/` while excluding the `Proxy` and `TestContracts` subdirectories.

Files Included

- `contracts/*`

Files Excluded

- `contracts/Proxy/*`
- `contracts/TestContracts/*`

Findings

ERD-1

EToken Transferability Leads to Potential for Collateralization Inaccuracy

• High ⓘ

Fixed

✓ Update

Marked as "Fixed" by the client. Addressed in: `87680502edc325f16b93658111635ba482b9a5eb` . The client provided the following explanation:

Disabled transferability of EToken.

File(s) affected: `EToken.sol` , `CollateralManager.sol`

Description: ETokens are transferable tokens that represent a borrower's collateral. For each different collateral type, there is a different type of EToken. Burning EToken is necessary to release the corresponding collateral. Moreover, when redeeming USDE and paying off a trove's debt, the trove's EToken also gets burned proportionately.

ETokens are transferable. Therefore, they may be sent to accounts that do not have open troves. If this is the case, then the collateral is essentially held in an account wherein it cannot be claimed by people redeeming USDE, as doing so requires redeeming from open troves. As a result, sending ETokens to accounts without open troves will lock up collateral indefinitely, creating a situation where that collateral is effectively no longer backing USDE.

While on its own this is not a dire issue, the problem is exacerbated by the fact that the TCR (Total Collateralization Ratio) is always calculated with respect to the total supply of the various ETokens. Every path to calculate TCR involves passing through `CollateralManager.getEntireCollValue(_price)` , where on lines 588-590, the amount of collateral that does not comprise pending liquidation rewards is calculated with respect to EToken supply. As a result, collateral that is no longer backing USDE will erroneously be included in the calculation of TCR, which will have systemic effects like affecting the interest rate and affecting the timing of when recovery mode is activated.

Recommendation: There are a few possible routes to resolve this problem:

1. Disable EToken transferability.
2. Prevent EToken transfers to those who do not have open troves.
3. Deduct from the active pool's effective balance if ETokens are transferred to non-borrower addresses, taking care to re-integrate those funds if the ETokens are once again sent to borrower addresses.

ERD-2

Changing USDE Gas Compensation Causes Discrepancy Between Trove ICRs and TCR

• High ⓘ

Fixed

✓ Update

Marked as "Fixed" by the client. Addressed in: `6c74c1716105a6b9bf25db1b19c35a6fc8a6a243` . The client provided the following explanation:

GasPool mint or burn when USDE_GAS_COMPENSATION be updated.

File(s) affected: `CollateralManager.sol`

Description: The function `setGasCompensation()` allows the owner of `CollateralManager` to set the value `USDE_GAS_COMPENSATION`, which is used to determine how much a borrower owes for compensating liquidators for calling liquidation functions. The value of `USDE_GAS_COMPENSATION` is used to determine, upon opening a trove, how much of the issued debt (in the form of USDE) is to be sent to the gas pool to compensate the liquidation of the trove.

Suppose the USDE gas compensation has been increased by calling the function. Now, via `TroveManager.getTroveDebt()` which calculates a trove's debt by summing up a certain amount with the *new* USDE gas compensation amount, the ICR of **all existing troves** is increased. However, the TCR of the protocol, which calculates the gas compensation portion by relying on the balance of the gas pool, will remain unchanged. The result will be a potentially massive discrepancy between the ICRs and the TCR, whereas in general one should be able to draw a direct formal relationship between the two. Potential consequences of this may include troves being put up for liquidation sooner or later than expected, recovery mode being triggered sooner or later than expected, and the TCR or ICRs of troves being materially misrepresented to users and devs.

Recommendation: A few routes could be taken to resolve this issue:

1. Prevent the possibility of changing USDE gas compensation.
2. Do not retroactively cause existing troves to have a different debt burden on account of changing the USDE gas compensation amount.
3. Mint to or burn from the gas pool when changing USDE gas compensation in order to make up for the discrepancy between the ICRs and TCR.

ERD-3 First Collateral Never Has Its Price Updated

• High ⓘ

Acknowledged

Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

ETH is the first supported collateral, and the price of ETH will be updated first before calling `priceUpdated()`.

File(s) affected: `CollateralManager.sol`

Description: `CollateralManager.priceUpdate()` iterates through the supported collaterals and fetches an updated price from their oracles. However, the for loop starts iterating from index `1`. Nothing in `addCollateral()` prevents any particular collateral type from occupying index `0`. Such a collateral would never have its price updated.

Recommendation: Update `priceUpdate()` to iterate through all collaterals. If the first entry is intended to be WETH, enforce this during initialization or in `addCollateral()`.

ERD-4

Liquidations Are Not Proportionally Funded by All Troves

• High ⓘ

Mitigated

Alert

We are marking this issue as "Mitigated". This is because although this fix does follow one of the recommendations, the result is that a low collateral ratio in troves holding one type of collateral can negatively impact the overall TCR of the protocol.

Addressed in: `be0a7597fbd9d2ee1d2e19fd4645b362dc05b781`. The client provided the following explanation:

Ensure that each collateral has at least one trove, and we will open a trove containing all the collateral.

File(s) affected: `TroveManager.sol`

Description: When active troves are liquidating other troves, assets are liquidated with respect to proportions of collateralized asset types. For example, for a trove that only holds rETH that gets liquidated, all of its rewards (collateral and debt) will go towards only troves that hold rETH. This could lead to scenarios where some troves take on a disproportionate amount of liquidated debt if they use a less popular collateral type. In the case of a particular collateral type taking a steep dive in price, this could lead to liquidation cascades. **At its most extreme, liquidating the last trove that holds a given collateral type may result in perpetually unclaimed debt.**

Recommendation: Consider funding liquidations with all troves, regardless of collateral type. Otherwise, explicate in documentation that this is an intentional design decision and discuss the above consequences. If maintaining this logic for liquidation, consider inserting a function that prevents the last trove of a given collateral type from being liquidated. Moreover, consider enforcing the existence of a trove that holds all collateral types and that cannot be closed.

ERD-5

Insufficient Means for Handling stETH Rebasing

• High ⓘ Acknowledged

Update

At the moment, shares and balance are equivalent. That is to say, shares are not currently being recorded.

Marked as "Acknowledged" by the client. The client provided the following explanation:

EToken will maintain the same rebasing mechanism as the corresponding collateral. For rebase tokens, the protocol records the share value.

File(s) affected: EToken.sol, TroveManager.sol, ActivePool.sol, DefaultPool.sol

Description: When a user deposits stETH into a trove they receive eTokens equivalent to the deposit amount. However, stETH rebases such that an address holding stETH will gradually receive more stETH over time. As of now, the codebase does not display any mechanism for allowing borrowers who collateralize stETH to claim these rewards. This will result in excess stETH accumulating in ActivePool and DefaultPool.

Recommendation: Fully handle stETH's rebasing mechanism in the contracts so that each user's stETH collateral rewards are correctly accounted for. We recommend implementing extensive testing on this front to ensure that everything is working as expected.

ERD-6

Flashloans Can Be Used to Siphon Liquidation Rewards Through Active Trove Liquidation Mechanism

• High ⓘ Acknowledged

Update

Despite the client's response, we are nonetheless concerned that the form of transaction enabled by this issue negatively impacts trove holders.

Marked as "Acknowledged" by the client. The client provided the following explanation:

We believe this is normal arbitrage behavior that is allowed.

File(s) affected: BorrowerOperations.sol, TroveManagerLiquidations.sol, TroveManager.sol, CollateralManager.sol

Description: A liquidator can open a trove using a flashloan to deposit a large amount of collateral, increasing the number of rewards per unit staked to which they are entitled. In the same transaction they can trigger a liquidation and receive a portion of the redistributed rewards. They can then close their trove, also within the same transaction.

This attack effectively steals a portion of the collateral rewards that should be going to existing trove holders.

Note that the effectiveness of this attack increases as the size of the liquidation increases and as the amount of the collateral in the system decreases (so that they can more easily get a larger share).

Exploit Scenario:

1. Attacker notices (a) large position(s) that may be liquidated.
2. Attacker sends a transaction performing the following steps:
 1. Take out a flash loan in the same collateral type that is being liquidated.
 2. Open a trove with the minimum debt and uses as much collateral as possible (this will ensure that his share count is high).
 3. Liquidate the trove(s) and thereby receive the redistributed collateral rewards.
 4. Close the trove.

Recommendation: Though this attack may only be feasible in specific circumstances, it may be beneficial to prevent a user from opening and closing a trove in the same block to prevent this from happening. This can be accomplished by adding `_updateCallerBlock()` to the beginning of `openTrove()` and `_checkSameTx()` to the beginning of `closeTrove()`.

```
function _updateCallerBlock() internal {
    _lastCallerBlock = keccak256(abi.encodePacked(tx.origin, block.number));
}

function _checkSameTx() internal view {
    require(keccak256(abi.encodePacked(tx.origin, block.number)) != _lastCallerBlock,
"8");
}
```

ERD-7

Oracle Manipulation with Flashloans Can Exploit Funds via Liquidation

• High ⓘ

Mitigated

Alert

Although this issue was addressed by Tellor, it was only done so for the case of ETH, not stETH and rETH.

Marked as "Acknowledged" by the client. The client provided the following explanation:

It has been fixed by Tellor: <https://www.liquity.org/blog/tellor-issue-and-fix>

File(s) affected: `StabilityPool.sol`, `PriceFeed.sol`

Description: The oracle contracts in ERD currently rely on Chainlink and Tellor oracles. Tellor is relied upon as a fallback oracle. However, Tellor's prices can be manipulated within the space of a block: reporting a price to Tellor only requires staking 10 TRB.

Therefore, if Chainlink is ever determined by the oracle contracts to be frozen or broken, then an attacker may be able to buy USDE using a flashloan, deposit it in the stability pool, report a false price to Tellor and thereby trigger recovery mode, enabling the liquidation of all troves with TCR below 130%; part of the gains can be sold off to repay the flashloan.

Exploit Scenario:

1. Chainlink oracle malfunctions.
2. Attacker borrows funds from flashloan and buys USDE;
3. Attacker deposits USDE into stability pool;
4. Attacker reports a price of a collateral token to be low enough such that the TCR of the protocol is low enough to trigger recovery mode;
5. Attacker liquidates all troves that they can thanks to recovery mode;
6. Attacker withdraws funds from stability pool and pays back flash loan.

Recommendation: Consider reverting in case a change of price via Tellor is what leads to the triggering of recovery mode. Alternatively, consider preventing a single transaction from both depositing to and withdrawing from the stability pool.

ERD-8

EToken Transferability May Lead to Undercollateralized Trove

• Medium ⓘ Fixed

✓ Update

Marked as "Fixed" by the client. Addressed in: `87680502edc325f16b93658111635ba482b9a5eb`. The client provided the following explanation:

Disabled transferability of EToken.

File(s) affected: `EToken.sol`, `CollateralManager.sol`

Description: ETokens require that the trove address from which they are being sent does not drop in ICR below the MCR (enforced by `EToken._requireValidAdjustment()`). However, in `CollateralManager.validAdjustment()`, where this condition is determined, only the prices of ETH and the collateral type being transferred are being updated (on lines 617-618). This means that the ICR and TCR calculations that are made to determine whether the trove adjustment is valid may be relying on stale data. For example, suppose that an rETH EToken is being transferred. In this case, although the prices of ETH and rETH may be updated, the price of stETH will not be updated. As a result, if the price of stETH dropped significantly since the last price update, the EToken transfer may be allowed to proceed despite the actual ICR of the trove decreasing below the MCR as a result.

Recommendation: Either require all collateral prices to be updated on EToken transfers or prevent EToken transfers entirely.

ERD-9

Redemptions and Liquidations Affect Troves in Incorrect Order

• Medium ⓘ Acknowledged

i Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

Acknowledge in documentation that trove sorting is done using old ICRs and that redemptions and liquidations are affected accordingly. And we will manually update ICR sorting as appropriate.

File(s) affected: `SortedTrove.sol`, `TroveManagerRedemptions.sol`, `TroveManager.sol`

Description: When a trove is inserted into `SortedTrove`, the ICR is recorded in the trove's corresponding `Node` struct as `Node.ICR`. However, as time passes, this recorded ICR is bound to become inaccurate with regard to the actual trove's ICR—the price of ETH is variable, and so is the amount of debt for which a trove is held responsible. Furthermore, the various collateral types are bound to fluctuate in price as well; notably, rETH is expected to increase in price with respect to time. Therefore, the sorting of the troves according to the stored ICR fields will change from that of a hypothetical sorted list that uses the actual ICR values.

It is worth bearing in mind that there exists the `TroveManagerRedemptions.updateTrove()` function that allows the caller to update the trove sorting by reinserting specified troves—each reinsertion involves recording the trove's freshly updated ICR. However, this imposes a significant gas burden for the lifetime of the protocol on whatever parties take on the responsibility of the updating the trove sorting. Moreover, when reinserting the troves, while the ICR of the reinserted trove will be fresh, its location will depend on the potentially stale ICRs recorded in `SortedTrove`. Therefore, keeping the troves in the correct order may require repeatedly updating the ICRs of ALL the nodes in the list. In terms of protocol security, improper sorting of the troves poses a problem because trove sorting is the basis for the correct ordering of liquidations and redemptions: for as long as the sorting remains incorrect, troves are liable to be redeemed against in the incorrect order—redemptions are made according to the trove ordering such that troves with lower ICRs (albeit above MCR) are redeemed

against first—and liquidations that are made with the sequential liquidation function may liquidate troves in the wrong order (and lower ICR troves may be skipped over).

It should be noted that documentation makes the following false statement as well:

Nodes also remain sorted as the ETH:USD price varies, since price fluctuations change the collateral value of each Trove by the same proportion.

Recommendation: We present three recommendations that can be implemented to alleviate this issue:

1. Do not store trove ICRs in SortedTrove. Instead, dynamically pull trove ICRs from TroveManager and use those to compare ICRs.
2. Do not allow sequential liquidations—require troves to be liquidated using the batch liquidation function. In the event that trove sorting is inaccurate, this will prevent the mistaken assumption that liquidating in sequence would occur according to actual ICR.
3. Acknowledge in documentation that trove sorting is done using old ICRs and that redemptions and liquidations are affected accordingly.

ERD-10 Maximum Borrow Rate Not Enforced

• Medium ⓘ Fixed

✓ Update

Marked as "Fixed" by the client. Addressed in: `295913f6e865e18e56e4d776f61209752a4eb48c`. The client provided the following explanation:

Borrow rate cannot exceed the value given by `getMaxBorrowRate()`.

File(s) affected: `TroveInterestRateStrategy.sol`

Description: ERD applies a variable interest rate in the calculation of outstanding debt. Per the function `getMaxBorrowRate()`, this amount should be capped at `baseBorrowRate + rateSlope1 + rateSlope2`. However, the function `calculateInterestRates()`, in the case where `vars.utilizationRate < optimalUtilizationRate`, returns `baseBorrowRate + rateSlope1 + rateSlope2 * remaindUtilizationRateRatio` where the value `remaindUtilizationRateRatio` may be greater than 1. This means that the returned value can be greater than the value returned by `getMaxBorrowRate()`.

Recommendation: Determine whether there is to be a maximum borrow rate. If yes, make sure that the values returned by `calculateInterestRates()` cannot exceed the value given by `getMaxBorrowRate()`. If no, consider removing the function `getMaxBorrowRate()`.

ERD-11

Upgrades Can Enable Unexpected Loss of Funds

• Medium ⓘ Acknowledged

i Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

The relevant permissions of the protocol configuration are given to the time lock contract, and the contract upgrade and emergency suspension permissions are given to the multi-signature wallet.

File(s) affected: `ActivePool.sol`, `DefaultPool.sol`, `EToken.sol`, `USDEToken.sol`

Description: All of the collateral in the protocol is stored between `ActivePool` and `DefaultPool`. Both contracts inherit from `OwnableUpgradeable`, which means that both can be upgraded by a user with a privileged role *immediately*. Moreover, both pools' funds can be accessed through other contracts inheriting from `OwnableUpgradeable`, which means if any of these other contracts' owners are compromised, they may also be able to siphon funds from the pools with great speed and ease. This is also possible through updating `EToken`, which is a token that can be used to redeem from troves.

The last possibility is also applicable to `USDEToken` ; if upgraded in a particular way (say, by letting an owner transfer the tokens indiscriminately), all the tokens can be stolen.

Recommendation: In general, it is ill-advised to keep public protocol funds in upgradeable contracts or contracts that can be significantly manipulated by privileged roles. However, if this must be the case, this must be stated in protocol docs. The same holds for token contracts.

Moreover, it would be useful to give admin powers only to timelock contracts. This way, even if a malicious user does assume control, there may be a notable window of opportunity for users to retrieve their funds before an attack is executed. If the ability to quickly respond to hacks remains a concern, such functions may be designated for instantaneous execution rather than strictly via timelock.

ERD-12 Privileged Roles and Ownership

• Medium ⓘ Acknowledged

i Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

It will be made clear to the users.

Description: Privileged roles are used extensively throughout the protocol for the purpose of configuration. However, we have identified a number of functions and roles that can lead to the loss of funds.

- `USDEToken` :
 - `troveManagerAddress` and `borrowerOperationsAddress` can call `mintToTreasury()` and send mint arbitrary amounts of USDE to the `treasuryAddress`.
 - `stabilityPoolAddress` and `troveManagerLiquidationsAddress` can transfer any amount of USDE tokens between any addresses.
- `ActivePool` : The `borrowerOperationsAddress`, `troveManagerAddress`, `stabilityPoolAddress`, `troveManagerRedemptionsAddress`, and `troveManagerLiquidationsAddress` can transfer all held collaterals to the `treasuryAddress` and `liquidityIncentiveAddress`. **All of these addresses can be set and changed by the contract owner.**
- `DefaultPool` : The `troveManagerAddress` can transfer all held collaterals to the `activePoolAddress`. **All of these addresses can be set and changed by the contract owner.**
- `LiquidityIncentive` : The owner has unrestricted access to transferring the contract's tokens.
- `StabilityPool` : `borrowerOperations` has unlimited approval for the contract's `WETH`. **The owner can set and change the `borrowerOperations` address as well as the `WETH` address to another ERC-20 token.**
- `Treasury` : The owner has unrestricted access to transferring the contract's tokens.

Recommendation: This centralization of power needs to be made clear to the users, especially depending on the level of privilege the contract allows to the owner. Consider also using a timelock contract for calling these functions.

ERD-13

Users Have to Pay Interest on the Borrowing Fee

• Medium ⓘ Acknowledged

i Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

The proportion of borrow fees is negligible.

File(s) affected: `BorrowerOperations.sol`

Description: Users are charged a borrowing fee that is added to their total USDE Debt, which accumulates interest. However, the borrowing fee is minted immediately to the treasury and is never received by the borrower. The borrowing fee should not accumulate interest as it is not loaned to the borrower.

Recommendation: Track the borrowing fee separately from the debt such that it does not accumulate interest.

ERD-14 USDE Debt May Be Misaccounted For

• Medium ⓘ

Acknowledged

Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

In `ActivePool`, `increaseDebt` is the principal amount of the debt, while `decreaseDebt` is the debt including compound interest. This has no impact on the protocol.

File(s) affected: `DefaultPool.sol`, `ActivePool.sol`

Description: Both the `DefaultPool` and `ActivePool` track their USDE debt. The function `increaseUSDEDebt()` will always increase the debt by the full `_amount`. The function `decreaseUSDEDebt()` will decrease its debt by the `min(_amount, USDEDebt)` to prevent underflow. These functions are often used to transfer debt from one pool to another. Given the asymmetry between `increaseUSDEDebt()` and `decreaseUSDEDebt()`, it is possible for a pool to increase its debt by a quantity more than the other pool decreased its debt. This would create a surplus of debt where none exists.

Recommendation: Update `decreaseUSDEDebt()` to return how much it was able to decrease the debt by. Pass this value to `increaseUSDEDebt()`.

ERD-15

Removing Collateral Types Prevents Collateral Withdrawal

• Medium ⓘ

Acknowledged

Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

The collateral must be in the pause state before being removed. In this state, the user can `withdrawColl` and `closeTrove`. Before removal, we will issue an announcement to users about the time to withdraw relevant collateral.

File(s) affected: `CollateralManager.sol`, `BorrowerOperations.sol`

Description: The function `CollateralManager.removeCollateral()` removes a collateral type from the list of supported collaterals. Once no longer supported, users cannot withdraw their collateral in `BorrowerOperations.withdrawCollateral()`. Any users with a non-zero EToken balance would have no way of reclaiming their collateral.

Additionally, if the EToken is changed by calling `CollateralManager.setEToken()`, the total supply of the previous EToken would map to irredeemable collateral.

Furthermore, removal of a collateral type could result in a very sudden drop of users' ICRs. To take an extreme example, if a user has deposited only stETH and it becomes unsupported, their ICR would drop to zero. This would have a corresponding effect on TCR as well.

Finally, if an EToken is transferred to an address without a trove, such users would be unable to redeem their collateral until they opened a trove.

Recommendation:

1. Either create a way for users to redeem their ETokens for unsupported collaterals, or disable `removeCollateral()` and `setEToken()` for collateral types that have a non-zero total supply of their EToken.
2. Create a redemption method for users to redeem their ETokens for now-unsupported collateral.

ERD-16

Removing Collateral Types Adversely Affects Collateralization Ratios

• Medium ⓘ

Acknowledged

Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

The same as [ERD-15](#).

File(s) affected: `CollateralManager.sol`

Description: The function `CollateralManager.removeCollateral()` removes a collateral type from the list of supported collaterals. Removal of a collateral type could result in a very sudden drop of users' ICRs. To take an extreme example, if a user has deposited only stETH and it becomes unsupported, their ICR would drop to zero. This would have a corresponding effect on TCR as well.

The consequent effects may include multiple troves being put up for liquidation simultaneously as well as triggering of recovery mode.

Recommendation: Consider the following options:

1. Disable collateral type removals, favoring contract migration instead.
2. Add guard checks that prevent individual troves and TCR being severely affected by collateral type removal.

ERD-17

Pending Debt Rewards Do Not Accrue Interest, Disincentivizing Users From Interacting with Protocol

• Medium ⓘ

Acknowledged

Update

We disagree with the reasoning provided in the client's explanation, as whether debt is allocated or pending allocation is a consequence of a gas-saving implementation detail.

Marked as "Acknowledged" by the client. The client provided the following explanation:

We think that this part of the debt has not been actually allocated to trove, so the interest should not be accumulated.

File(s) affected: `TroveManager.sol`

Description: When liquidating a trove, one possibility is for the debt and collateral of that trove to be liquidated across all remaining active troves. This is done by adding the funds to the `DefaultPool` and applying the debt and collateral to an applicable trove (via `TroveManager.applyPendingRewards()`) when it is being adjusted, when it is being closed, and when it is being redeemed against. However, while debt is held in `DefaultPool`, interest is not accrued against it as it does not count towards the debt of an active trove.

As a result, it would be in a borrower's interest to, with all else being equal, wait for as long as possible to adjust a trove or close it, as doing so would allow them to avoid paying interest on pending debt rewards.

Recommendation: Consider having debt accrue on USDE that is in `DefaultPool`.

ERD-18 Unchecked Return Value

• Medium ⓘ

Acknowledged

Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

These return values will not have a bad impact on the protocol for the time being.

File(s) affected: BorrowerOperations.sol , CollateralManager.sol , StETHOracle.sol , PriceFeed.sol , StabilityPool.sol , TroveManager.sol , TroveManagerRedemptions.sol

Related Issue(s): SWC-104

Description: Not checking return values can lead to unintended consequences, especially when the return value indicates the success of a process. Consider the following instances where return values are unchecked:

- BorrowerOperations.sol#257
- BorrowerOperations.sol#723
- BorrowerOperations.sol#869
- BorrowerOperations.sol#871
- CollateralManager.sol#249
- CollateralManager.sol#618
- StETHOracle.sol#276-282
- StETHOracle.sol#285-301
- StETHOracle.sol#314-331
- PriceFeed.sol#613-628
- PriceFeed.sol#637-643
- PriceFeed.sol#646-662
- PriceFeed.sol#674-691
- StabilityPool.sol#276
- StabilityPool.sol#1271
- TroveManager.sol#392-396
- TroveManager.sol#1270
- TroveManagerRedemptions.sol#214
- EToken.sol#104
- USDEToken.sol#192

Recommendation: Check the return values.

ERD-19

Approximate Hints are Generated Sub-Optimally

• Low ⓘ Acknowledged

Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

We knew it.

File(s) affected: HintHelpers.sol , SortedTrove.sol

Description: HintHelpers.getApproxHint() attempts to generate an approximate hint for the proper placement of a trove in the SortedTrove list during insertion. However, it does this based on the ICRs of the troves in SortedTrove using price data from the latest oracles' updates. On the other hand, the insertion of the troves into SortedTrove is dependent on the ICR of these troves at the time of insertion, and this quantity may be very stale, potentially going back to the time of the first insertion. Therefore, as ICR is bound to change, getApproxHint() is not using the best means of discerning a good hint.

Recommendation: One of the two following recommendations may resolve the issue:

1. When calculating a hint, instead of calculating the ICRs of the troves in SortedTrove using TroveManager.getCurrentICR() , instead use the ICR values stored in SortedTrove .
2. Switch from using stored ICR values in SortedTrove to retrieving them from TroveManager when inserting.

ERD-20 Missing Input Validation

• Low ⓘ

Fixed

✓ Update

Marked as "Fixed" by the client. Addressed in: `381e9951d6371e2b1dab4dd1bfd75c7a136fdb54` . The client provided the following explanation:

`BorrowerOperations._adjustTrove()` validated before calling it. `SortedTrove/USDEToken` already be validated before calling it.

File(s) affected: `TroveInterestRateStrategy.sol` , `EToken.sol`

Description: Below are a number of instances where we identified missing input validation.

- `BorrowerOperations._adjustTrove()` : Validate that the entries in `_params.amountIn[]` are non-zero.
- `CollateralManager.addCollateral()` : Validate that the input addresses are non-zero.
- `CollateralManager.setOracle()` : Validate that the input addresses are non-zero.
- `CollateralManager.setEToken()` : Validate that the input addresses are non-zero.
- `CollateralManager.setMinDebt()` : Validate that `_minDebt` is greater than or equal to `USDE_GAS_COMPENSATION` .
- `CollSurplusPool.setAddresses()` : Validate that `_collateralManagerAddress` is a non-zero address.
- `EToken.setAddresses()` : Validate that the input addresses are non-zero.
- `TroveInterestRateStrategy.setAddresses()` : Validate that the input addresses are non-zero.
- `SortedTrove.getNext()` : Require that `data.nodes[_id].exists == true` . Otherwise `_id` may be misinterpreted to be the tail.
- `SortedTrove.getPrev()` : Require that `data.nodes[_id].exists == true` . Otherwise `_id` may be misinterpreted to be the head.
- `USDEToken.sendToPool()` and `USDEToken.returnFromPool()` : require that `_poolAddress` is a valid pool address.

Recommendation: Add the missing input validation.

ERD-21 Implementation Contracts Can Be Initialized

• Low ⓘ

Fixed

✓ Update

Marked as "Fixed" by the client. Addressed in: `7717c836eaa4b1c32d76126e1bf50e33fc29c0d5` .

Description: The implementation contracts behind a proxy can be initialized by any address. This is not a security problem in the sense that it impacts the system directly, as the attacker will not be able to cause any contract to self-destruct or modify any values in the proxy contracts. However, taking ownership of implementation contracts can open other attack vectors, like social engineering or phishing attacks.

Recommendation: Call `_disableInitializers()` in constructors of upgradeable contracts to avoid the initialization of the implementation contracts.

ERD-22

Price Feed May Unnecessarily Resort to Stale Data

• Low ⓘ

Acknowledged

i Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

We use the oracle that liquidity used.

File(s) affected: `PriceFeed.sol`

Description: At `PriceFeed.sol#265`, it is required that both Chainlink and Tellor are working in order for the price to be updated. This means that if Chainlink is restored while Tellor is not, the feed will continue to use the "last good price" instead of proceeding with the latest Chainlink price. The vice versa case is also true: if Tellor is restored while Chainlink is not, the feed will continue to use the "last good price" instead of proceeding with the latest Tellor price. Hence stale data will end up being preferred to fresh data.

A similar situation occurs at `PriceFeed.sol#234`, where if Chainlink becomes operational at the same time as Tellor becomes unreliable, fresh Chainlink data will not be used.

Recommendation: Rework the logic so that fresh data is always used when it can reliably be sourced from at least Chainlink.

ERD-23

When Oracles Are Down, Stale Prices Are Used

• Low ⓘ

Acknowledged

Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

We use the oracle that liquidity used.

File(s) affected: `PriceFeed.sol`

Description: In every case where `fetchPrice()` is called, either a price is pulled from an oracle or `lastGoodPrice` is called. Hence if both oracles are down, only the last good price can be resorted to. However, this means that even if both oracles are down or broken for a significant period of time, the last good price will be used indefinitely. This may result in unfavorable conditions for the protocol.

Recommendation: Consider implementing a `revert` clause for the scenario where the last good price is excessively stale.

ERD-24 Contracts Lack Consolidated Initialization

• Low ⓘ

Acknowledged

Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

We knew it.

Description: Many contracts have both an `initialize()` function as well as a `setAddresses()` function. Almost all of the variable initialization takes place in `setAddresses()` for most contracts. Given that `setAddresses()` can be called more than once, all addresses can be reassigned at the owners discretion. Additionally, some contracts contain an additional `init()` function that initializes variables not assigned in either `initialize()` or in `setAddresses()`. The ability to reassign all variables increases the attack surface in the event of a compromised owner.

Recommendation: Consolidate all variable initialization into the `initialize()` function. If any addresses must support the ability to be reassigned, create a separate setter function for that specific address.

ERD-25

`MIN_NET_DEBT` Can Be Set Lower than `USDE_GAS_COMPENSATION`

• Low ⓘ

Fixed

Update

Marked as "Fixed" by the client. Addressed in: `381e9951d6371e2b1dab4dd1bfd75c7a136fdb54`.

File(s) affected: `CollateralManager.sol`

Description: There is no restriction on how low the `MIN_NET_DEBT` can be set by calling `CollateralManager.setMinDebt()`. If the `MIN_NET_DEBT` is set lower than `USDE_GAS_COMPENSATION`, a trove could be adjusted to have inadequate gas compensation.

Recommendation: Add a check that prevents `MIN_NET_DEBT` from being set to a quantity below `USDE_GAS_COMPENSATION`.

ERD-26

Loss of Precision Due to Division before Multiplication

• Low ⓘ

Acknowledged

Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

There is always a multiplication before division, which can basically meet the precision requirements of the protocol.

File(s) affected: `CollateralManager.sol`, `TroveManagerLiquidations.sol`

Description: Division before multiplication may result in a loss of precision when the operations are carried over integer numbers. `CollateralManager._calcValue()` contains an example of this at lines 302-309.

This issue is found in the following instances:

- `CollateralManager.sol#302-309`
- `TroveManagerLiquidations.sol#448-450`
- `TroveManagerLiquidations.sol#544-546`

Recommendation: Rewrite equations so that division happens after multiplication.

ERD-27

`EToken.transferFrom()` May Transfer an Incorrect Amount of Tokens

• Informational ⓘ

Fixed

Update

Marked as "Fixed" by the client. Addressed in: `87680502edc325f16b93658111635ba482b9a5eb`. The client provided the following explanation:

Disabled transferability of `EToken`.

File(s) affected: `EToken.sol`

Description: `EToken.transferFrom()` converts the `_amount` to `share` by calling `share = getAmount(_amount)`. Although `getShare()` and `getAmount()` currently implement the same logic, this may result in transferring an unexpected quantity of tokens if the logic of `getAmount()` ever differs from `getShare()` in a future revision.

Recommendation: Update the assignment to `share = getShare(_amount)`.

ERD-28 Ownership Can Be Renounced

• Informational ⓘ

Mitigated

Update

We are marking this as "Mitigated" as not all files were adjusted according to the recommendation.

Addressed in: `207dbda80ec51a389c16e31d41373e770297c7fd`.

Description: Contracts inheriting from `Ownable` or `OwnableUpgradeable` contain a function called `renounceOwnership()` which allows the owner to leave the contract without an owner. This would leave every function guarded by the `onlyOwner` modifier unable to be executed.

Recommendation: Confirm whether this is the intended behavior. If not, override the `renounceOwnership()` function to always revert.

ERD-29 Missing Initializers

• Informational ⓘ Fixed

✓ Update

Marked as "Fixed" by the client. Addressed in: `b6c7ac95cdb50db0bcac7cbd85e68458ce90d7de`.

File(s) affected: `BorrowerOperations.sol`, `StabilityPool.sol`, `TroveManager.sol`

Description: The affected contracts inherit from `ReentrancyGuardUpgradeable`. However, at no point is an initialization called for this parent contract from within the affected contracts.

Recommendation: Take care to call `__ReentrancyGuard_init()` or `__ReentrancyGuard_init_unchained()` in the `initialize()` functions of the affected contracts.

ERD-30 Redundant Code

• Informational ⓘ Acknowledged

i Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

This function is mainly for rebase tokens.

Description: When dealing with collateral redemption in the form of eTokens, the protocol often calls `CollateralManger.getShares()` which in turn calls `EToken.getShare()` however, this function simply returns the input, making it a redundant function. This in turn complicates the logic within certain function. Take, for example, the `CollSurplusPool.accountSurplus()` function, which contains the following section:

```
address[] memory collaterals = collateralManager.getCollateralSupport();
uint256[] memory shares = collateralManager.getShares(
    collaterals,
    _amount
);
```

The conversion of `_amount` to shares can be entirely removed since this in effect just returns the same array.

Furthermore functions such as `getShares()` and `getShare()` in the `CollateralManager` contract can be entirely removed.

Recommendation: Consider removing the redundant code.

ERD-31 Interest Rate Increases as Total Collateralization Rate Rises

• Undetermined ⓘ Acknowledged

i Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

If market conditions or user profiles change, we will revise the interest rate policy in order to improve capital efficiency.

Description: One of the design decisions made for this protocol was to have the variable interest rate increase as the total collateralization rate (TCR) of the protocol increases. We have decided to list this as an issue because in certain market conditions, this may create a positive feedback loop: depositors refuse to withdraw USDE because interest rates are high, leading to even more depositors refusing to withdraw USDE.

We acknowledge that the disadvantages of such a setup may not be so severe depending on the following conditions:

1. The majority of USDE will be withdrawn by well-funded borrowers, which will allow them to noticeably influence the interest rate;
2. The typical interest rate is expected to be competitive with regard to interest rates in traditional finance.

Recommendation: If market conditions or user profiles change, consider revising the interest rate policy in order to improve capital efficiency.

ERD-32

Frontend Functionality May Be Superfluous for ERD

• Undetermined ⓘ Acknowledged

Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

We hope to retain the functions of the decentralized front-end.

File(s) affected: `StabilityPool.sol`

Description: The frontend functionality in the `StabilityPool` contract with functions such as `registerFrontend()`, `getCompoundedFrontEndStake()`, `_updateFrontEndStakeAndSnapshots` is designed to provide a monetization mechanism for frontends that receive a large amount of user interaction.

If the ERD protocol does not intend for their to be a marketplace of front-ends then this functionality is superfluous and may even work against the goals of the developer team.

Recommendation: Evaluate the team's stance on having multiple independent frontends as opposed to only a single official frontend. If the intention is the latter, then eliminate all features that have to do with registering frontends.

ERD-33

Redemption of Troves with Lower ICR Creates Incentive for Increasing TCR

• Undetermined ⓘ Acknowledged

Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

It is forked from Liquity.

Description: The trove with the lowest ICR can be redeemed by any user. This creates an incentive for every participant that wants to keep their position open from being the trove with the lowest ICR. This can cause a race to the bottom dynamic which may negate or undermine the incentives created to keep troves as capital efficient as possible.

Exploit Scenario: Consider a case with two users A and B that want to keep their position open. Suppose $ICR(A) < ICR(B)$, then A has an incentive to increase their ICR to the point where $ICR(B) < ICR(A)$. But now B has the same incentive, and the process repeats indefinitely.

Recommendation: If this is undesired, consider having redemptions affect all troves uniformly, regardless of ICR.

Definitions

- **High severity** – High-severity issues usually put a large number of users' sensitive information at risk, or are reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
- **Medium severity** – Medium-severity issues tend to put a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or are reasonably likely to lead to moderate financial impact.
- **Low severity** – The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
- **Informational** – The issue does not post an immediate risk, but is relevant to security best practices or Defence in Depth.
- **Undetermined** – The impact of the issue is uncertain.
- **Fixed** – Adjusted program implementation, requirements or constraints to eliminate the risk.
- **Mitigated** – Implemented actions to minimize the impact or likelihood of the risk.
- **Acknowledged** – The issue remains in the code but is a result of an intentional business or design decision. As such, it is supposed to be addressed outside the programmatic means, such as: 1) comments, documentation, README, FAQ; 2) business processes; 3) analyses showing that the issue shall have no negative consequences in practice (e.g., gas analysis, deployment settings).

Code Documentation

1. Many `external` functions are missing NatSpec.
2. The documentation to `TroveLogic.updateState()` should document an additional purpose: minting accumulated interest to the treasury.
3. Incomplete references to documentation and/or references to external project documentation:
 1. `StabilityPool.sol#52`
 2. `StabilityPool.sol#133`
 3. `StabilityPool.sol#146`
4. There are many incorrect references to NICR where ICR is currently being used in implementation.

Adherence to Best Practices

1. Contracts using `SafeMathUpgradeable` are doing so unnecessarily, as the Solidity compilers version 0.8 and above check for overflows and underflows.
2. `++i` can be used instead of `i++` in for loops in order to save gas (look for instances of `j++` too). Consider combining this with `unchecked{}` blocks for even greater savings.
3. Instead of using `require()`, custom errors can be used to save gas.
4. `BorrowerOperations.sol#219` contains an unnecessary declaration.
5. In the flow of `TroveManager.decayBaseRateFromBorrowing()`, the calculation `block.timestamp.sub(lastFeeOperationTime)` is performed twice. Consider refactoring to perform this calculation only once in order to save gas.
6. Avoid using `assert()` in production code. Use `require()` or custom errors instead.
7. Remove commented out code:
 1. `ERDBase.sol#24`
 2. `ERDBase.sol#71-77`
 3. `EToken.sol#113`
 4. `EToken.sol#118`

8. Remain consistent with how conditions are checked. For example:
 1. `_requireTroveIsActive()` is used in `_adjustTrove()` but `require()` is used directly to check the same condition.
9. Correct typos:
 1. `TroveManager.hasPeningRewards()`
10. Instead of chaining external calls, consider directly calling the contract with the important information:
 1. `BorrowerOperations.sol#576` -- calling `TroveManager` instead of `CollateralManager` directly
11. Unnecessary code:
 1. `else continue;` at `TroveManagerLiquidations.sol#1072`
12. Prices are being fetched unnecessarily often, consuming excess gas. Consider cleaning up the code in order to reduce gas consumption. For example, in `TroveManagerRedemptions.updateTrove()`, prices are unnecessarily fetched at:
 1. `TroveManagerRedemptions.sol#671`
 2. `TroveManagerRedemptions.sol#672`
13. At `StabilityPool.withdrawCollateralGainToTrove()`, we see that `getDepositorCollateralGain()` is called twice, leading to unnecessary gas expenditure. These calls occur at lines 444 and 453.
14. At `TroveInterestRateStrategy.sol#130`, the total debt of the protocol is being calculated in order to later use whether it is zero as a condition. However, whether the amount is zero or not does not depend on `troveData.borrowIndex` being part of the calculation. Therefore, it would save gas to remove `.rayMul(troveData.borrowIndex);` from the calculation while preserving the original utility.
15. `ERDSafeMath128.sol` is superfluous as overflow checks are done automatically in Solidity 0.8.
16. The check at `BorrowerOperations.sol#646-649` is unnecessary due to the overflow check at lines 641-643.
17. `ActivePool._requireCallerIsBOrTroveMorSP()`: Update the function name to reflect the allowance of `TroveManagerRedemptions` and `TroveManagerLiquidations`.
18. `USDEToken._requireCallerIsBOrTroveMorSP()`: Update the function name to reflect the allowance of `TroveManagerRedemptions`.
19. `USDEToken._requireValidReceipient()`: Add `TroveManagerLiquidations` and `TroveManagerRedemptions` to the list of prohibited receivers.
20. `USDEToken.burn()`: Gives unnecessary access to `TroveManager`.
21. Trove statuses should always be represented by Enums and not raw numbers (see for example line 209 of `BorrowerOperations`, or the `getIsActive()` and `getIsSupport()` functions of `CollateralManager`).
22. Change the name of the `setAddress()` function to `setTroveManager()` in the Trove debt contract to clarify the purpose of the function.
23. The assignment `gas = USDE_GAS_COMPENSATION()` in the `TroveManagerRedemptions._redeemCollateralFromTrove()` function can occur earlier to avoid making two calls to `USDE_GAS_COMPENSATION()`.
24. In the `SortedTrove.findInsertPosition()` there is a redundant `else if` statement that can be skipped:

```
    } else if (nextId == address(0)) {
        // No `nextId` for hint – descend list starting from `prevId`
        return _descendList(_ICR, prevId);
    } else {
        // Descend list starting from `prevId`
        return _descendList(_ICR, prevId);
    }
```

Appendix

File Signatures

The following are the SHA-256 hashes of the reviewed files. A file with a different SHA-256 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different SHA-256 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

Contracts

- 148...071 ./contracts/USDEToken.sol
- 0bd...c4e ./contracts/HintHelpers.sol
- 2b9...908 ./contracts/TroveDebt.sol

- 670...b48 ./contracts/SortedTroves.sol
- 227...52f ./contracts/TroveInterestRateStrategy.sol
- 9e6...41c ./contracts/Errors.sol
- cc8...b65 ./contracts/MultiTroveGetter.sol
- 8f5...a47 ./contracts/TroveManagerLiquidations.sol
- 375...d31 ./contracts/CollateralManager.sol
- 048...667 ./contracts/EToken.sol
- bed...312 ./contracts/GasPool.sol
- 199...380 ./contracts/CollSurplusPool.sol
- 197...48a ./contracts/TroveManagerDataTypes.sol
- 3bc...c0f ./contracts/DataTypes.sol
- d69...534 ./contracts/DefaultPool.sol
- a1c...e6e ./contracts/TroveLogic.sol
- f02...584 ./contracts/StabilityPool.sol
- 732...5f6 ./contracts/ActivePool.sol
- f1b...786 ./contracts/TroveManagerRedemptions.sol
- 3a5...af6 ./contracts/LiquidityIncentive.sol
- c78...6b5 ./contracts/BorrowerOperations.sol
- dea...fb6 ./contracts/TroveManager.sol
- 018...525 ./contracts/CommunityIssuance.sol
- e7e...fe1 ./contracts/PriceFeed.sol
- 5bf...ef1 ./contracts/Treasury.sol
- 3b5...a75 ./contracts/Interfaces/ISortedTroves.sol
- dc2...1d8 ./contracts/Interfaces/IBorrowerOperations.sol
- a57...26b ./contracts/Interfaces/ITroveManagerLiquidations.sol
- c80...859 ./contracts/Interfaces/IERDBase.sol
- 0d9...151 ./contracts/Interfaces/IStETH.sol
- 789...4f0 ./contracts/Interfaces/ICollSurplusPool.sol
- 0a7...87e ./contracts/Interfaces/IEToken.sol
- 26a...248 ./contracts/Interfaces/IUSDEToken.sol
- 54b...74f ./contracts/Interfaces/IActivePool.sol
- 696...682 ./contracts/Interfaces/ICommunityIssuance.sol
- 1c2...a54 ./contracts/Interfaces/ITroveInterestRateStrategy.sol
- af9...0c4 ./contracts/Interfaces/ITroveDebt.sol
- 291...af2 ./contracts/Interfaces/IWETH.sol
- 2f4...75c ./contracts/Interfaces/IOracle.sol
- e5b...d0a ./contracts/Interfaces/IPool.sol
- fd9...6b1 ./contracts/Interfaces/IPriceFeed.sol
- ef6...260 ./contracts/Interfaces/ITroveManagerRedemptions.sol
- 256...1fb ./contracts/Interfaces/ITellorCaller.sol
- 6b3...906 ./contracts/Interfaces/ICollateralManager.sol
- ff4...a6a ./contracts/Interfaces/IStabilityPool.sol
- 632...ef3 ./contracts/Interfaces/IDefaultPool.sol
- 24e...8f8 ./contracts/Interfaces/ITroveManager.sol
- 465...fb8 ./contracts/Interfaces/ICollateralReceiver.sol
- 391...72d ./contracts/Dependencies/ERDBase.sol
- 8e2...44f ./contracts/Dependencies/BaseMath.sol

- 7bd...33a ./contracts/Dependencies/ITellor.sol
- 7e3...7d2 ./contracts/Dependencies/ERDMath.sol
- 404...68e ./contracts/Dependencies/AggregatorV3Interface.sol
- 021...7cd ./contracts/Dependencies/WadRayMath.sol
- 683...289 ./contracts/Dependencies/ERDSafeMath128.sol
- 734...de9 ./contracts/Dependencies/TellorCaller.sol
- b7a...4ca ./contracts/Oracles/StETHOracle.sol

Tests

- e1b...72e ./test/newBorrowerOperationsTest2.js
- 738...709 ./test/FeeArithmeticTest.js
- 1d4...cc8 ./test/BorrowerOperationsTest.js
- 37c...884 ./test/TroveManagerTest.js
- 10f...49b ./test/ERDSafeMath128Test.js
- e16...613 ./test/HintHelpers_getApproxHintTest.js
- 37f...d66 ./test/AccessControlTest.js
- 689...8ca ./test/StabilityPool_SPWithdrawalTest.js
- 801...051 ./test/MultiCollateralStabilityPoolTest.js
- dfd...13e ./test/StabilityPool_SPWithdrawalToCDPTest.js
- 0e5...206 ./test/BorrowerOperationsTestMultiCollateral.js
- 6db...3ef ./test/GasCompensationTest.js
- 894...847 ./test/USDETokenTest.js
- cef...06c ./test/CollSurplusPool.js
- 6e3...c40 ./test/CollateralManagerTest.js
- eda...954 ./test/StabilityPool_RoundingErrors.js
- 6f9...89d ./test/DefaultPoolTest.js
- f99...a0c ./test/TroveManager_RecoveryMode_Batch_Liquidation_Test.js
- b3e...9f8 ./test/PriceFeedTest.js
- 3cf...e2d ./test/SortedTroveTest.js
- dfe...a22 ./test/StabilityPoolTest.js
- 9c8...6e6 ./test/ERDMathTest.js
- 358...8c1 ./test/PoolsTest.js
- 7a1...989 ./test/ConnectContractsTest.js
- 0f1...9d7 ./test/TroveManager_RecoveryModeTest.js
- be7...23a ./test/TroveManager_LiquidationRewardsTest.js
- d2b...358 ./test/newBorrowerOperationsTest.js

Toolset

The notes below outline the setup and steps performed in the process of this audit.

Setup

Tool Setup:

- [Slither](#)  v0.9.2

Steps taken to run the tools:

1. Install the Slither tool: `pip3 install slither-analyzer`

2. Run Slither from the project directory: `slither .`

Automated Analysis

Slither

The following issues were detected by Slither and integrated into the report:

- unchecked return values;
- precision loss.

Test Suite Results

As of the initial audit commit, `npm hardhat test` results in the majority of tests failing.

While the test suite is extensive, the tests seem to be the least well-developed when it comes to features specific to ERD. More testing in the direction of alternative collaterals and interest accrual is recommended.

Update 1: The testing situation has not improved. Even fewer tests are being executed now, with most failing.

Update 2: Tests are no longer failing.

Compiled 104 Solidity files successfully (evm targets: paris, unknown evm version for solc version 0.4.23).

```
Contract: Access Control: ERD functions with the caller restricted to ERD contract(s)
  TroveManager
    ✓ applyPendingRewards(): reverts when called by an account that is not
BorrowerOperations or TMR (127ms)
    ✓ batchLiquidateTrove(): reverts when called by an account that is not TroveManger
(380ms)
    ✓ updateRewardSnapshots(): reverts when called by an account that is not
BorrowerOperations (111ms)
    ✓ removeStake(): reverts when called by an account that is not BorrowerOperations or
TMR (114ms)
    ✓ updateStakeAndTotalStakes(): reverts when called by an account that is not
BorrowerOperations
    ✓ closeTrove(): reverts when called by an account that is not BorrowerOperations
    ✓ addTroveOwnerToArray(): reverts when called by an account that is not
BorrowerOperations
    ✓ setTroveStatus(): reverts when called by an account that is not BorrowerOperations
    ✓ increaseTroveDebt(): reverts when called by an account that is not
BorrowerOperations
    ✓ decreaseTroveDebt(): reverts when called by an account that is not
BorrowerOperations/TMR
  ActivePool
    ✓ sendCollateral(): reverts when called by an account that is not BO nor TroveM nor
SP
    ✓ increaseUSDEDebt(): reverts when called by an account that is not BO nor TroveM
    ✓ decreaseUSDEDebt(): reverts when called by an account that is not BO nor TroveM
nor SP
  DefaultPool
    ✓ sendCollateralToActivePool(): reverts when called by an account that is not
TroveManager
    ✓ increaseUSDEDebt(): reverts when called by an account that is not TroveManager
    ✓ decreaseUSDE(): reverts when called by an account that is not TroveManager
  StabilityPool
    ✓ offset(): reverts when called by an account that is not TroveManagerLiquidations
```


USDEToken

- ✓ mint(): reverts when called by an account that is not BorrowerOperations
- ✓ burn(): reverts when called by an account that is not BO nor TroveM nor SP
- ✓ sendToPool(): reverts when called by an account that is not StabilityPool
- ✓ returnFromPool(): reverts when called by an account that is not

TroveManagerLiquidations nor StabilityPool

SortedTrove

- ✓ insert(): reverts when called by an account that is not BorrowerOps or TroveM
- ✓ remove(): reverts when called by an account that is not TroveManager
- ✓ reinsert(): reverts when called by an account that is neither BorrowerOps nor

TroveManager

CommunityIssuance

- ✓ trigger(): nothing
- ✓ issue(): nothing

CollateralManager

- ✓ pauseCollateral(): reverts when caller is not the Owner
- ✓ removeCollateral(): reverts when caller is not the Owner
- ✓ activeCollateral(): reverts when caller is not the Owner
- ✓ setOracle(): reverts when caller is not the Owner
- ✓ setCollateralPriority(): reverts when caller is not the Owner

TroveDebt

- ✓ addDebt(): reverts when caller is not the troveManager
- ✓ subDebt(): reverts when caller is not the troveManager

Contract: BorrowerOperations

Without proxy

- ✓ addColl(): reverts when top-up would leave trove with ICR < MCR (540ms)
- ✓ addColl(): Increases the activePool ETH and raw ether balance by correct amount

(380ms)

- ✓ addColl(), active Trove: adds the correct collateral amount to the Trove (378ms)
- ✓ addColl(), active Trove: Trove is in sortedList before and after (388ms)
- ✓ addColl(), active Trove: updates the stake and updates the total stakes (366ms)
- ✓ addColl(), active Trove: applies pending rewards and updates user's E_ETH,

E_USDEDebt snapshots (1590ms)

- ✓ addColl(), active Trove: adds the right corrected stake after liquidations have occurred (1029ms)

- ✓ addColl(), reverts if trove is non-existent or closed (834ms)

- ✓ addColl(): can add collateral in Recovery Mode (414ms)

- ✓ withdrawColl(): reverts when withdrawal would leave trove with ICR < MCR (477ms)

- ✓ withdrawColl(): reverts when calling address does not have active trove (505ms)

- ✓ withdrawColl(): reverts when system is in Recovery Mode (590ms)

- ✓ withdrawColl(): reverts when requested ETH withdrawal is > the trove's collateral

(728ms)

- ✓ withdrawColl(): reverts when withdrawal would bring the user's ICR < MCR (395ms)

- ✓ withdrawColl(): reverts if system is in Recovery Mode (551ms)

- ✓ withdrawColl(): doesn't allow a user to completely withdraw all collateral from

their Trove (due to gas compensation) (444ms)

- ✓ withdrawColl(): leaves the Trove active when the user withdraws less than all the collateral (422ms)

- ✓ withdrawColl(): reduces the Trove's collateral by the correct amount (357ms)

- ✓ withdrawColl(): reduces ActivePool ETH and raw ether by correct amount (348ms)

- ✓ withdrawColl(): updates the stake and updates the total stakes (349ms)

- ✓ withdrawColl(): sends the correct amount of ETH to the user (330ms)

- ✓ withdrawColl(): applies pending rewards and updates user's E_Coll, E_USDEDebt

snapshots (1383ms)

- ✓ withdrawUSDE(): reverts when withdrawal would leave trove with ICR < MCR (453ms)

- ✓ withdrawUSDE(): decays a non-zero base rate (1202ms)

- ✓ withdrawUSDE(): reverts if max fee > 100% (694ms)

- ✓ withdrawUSDE(): reverts if max fee < 0.25% in Normal mode (860ms)

- ✓ withdrawUSDE(): reverts if fee exceeds max fee percentage (993ms)

- ✓ withdrawUSDE(): succeeds when fee is less than max fee percentage (1611ms)

- ✓ withdrawUSDE(): doesn't change base rate if it is already zero (1299ms)
- ✓ withdrawUSDE(): lastFeeOpTime doesn't update if less time than decay interval has passed since the last fee operation (977ms)
- ✓ withdrawUSDE(): borrower can't grief the baseRate and stop it decaying by issuing debt at higher frequency than the decay granularity (1056ms)
- ✓ withdrawUSDE(): borrowing at non-zero base records the (drawn debt + fee) on the Trove struct (1026ms)
- ✓ withdrawUSDE(): Borrowing at non-zero base rate sends requested amount to the user (1021ms)
- ✓ withdrawUSDE(): Borrowing at zero base rate sends debt request to user (1079ms)
- ✓ withdrawUSDE(): reverts when calling address does not have active trove (514ms)
- ✓ withdrawUSDE(): reverts when requested withdrawal amount is zero USDE (574ms)
- ✓ withdrawUSDE(): reverts when system is in Recovery Mode (752ms)
- ✓ withdrawUSDE(): reverts when withdrawal would bring the trove's ICR < MCR (495ms)
- ✓ withdrawUSDE(): reverts when a withdrawal would cause the TCR of the system to fall below the CCR (448ms)
- ✓ withdrawUSDE(): reverts if system is in Recovery Mode (431ms)
- ✓ withdrawUSDE(): increases the Trove's USDE debt by the correct amount (381ms)
- ✓ withdrawUSDE(): increases USDE debt in ActivePool by correct amount (358ms)
- ✓ withdrawUSDE(): increases user USDEToken balance by correct amount (419ms)
- ✓ repayUSDE(): reverts when repayment would leave trove with ICR < MCR (435ms)
- ✓ repayUSDE(): Succeeds when it would leave trove with net debt >= minimum net debt (602ms)
- ✓ repayUSDE(): reverts when it would leave trove with net debt < minimum net debt (234ms)
- ✓ repayUSDE(): reverts when calling address does not have active trove (497ms)
- ✓ repayUSDE(): reverts when attempted repayment is > the debt of the trove (669ms)
- ✓ repayUSDE(): reduces the Trove's USDE debt by the correct amount (524ms)
- ✓ repayUSDE(): decreases USDE debt in ActivePool by correct amount (574ms)
- ✓ repayUSDE(): decreases user USDEToken balance by correct amount (517ms)
- ✓ repayUSDE(): can repay debt in Recovery Mode (550ms)
- ✓ repayUSDE(): Reverts if borrower has insufficient USDE balance to cover his debt repayment (422ms)
- ✓ adjustTrove(): reverts when adjustment would leave trove with ICR < MCR (491ms)
- ✓ adjustTrove(): reverts if max fee < 0.25% in Normal mode (509ms)
- ✓ adjustTrove(): decays a non-zero base rate (1331ms)
- ✓ adjustTrove(): doesn't decay a non-zero base rate when user issues 0 debt (1059ms)
- ✓ adjustTrove(): doesn't change base rate if it is already zero (663ms)
- ✓ adjustTrove(): lastFeeOpTime doesn't update if less time than decay interval has passed since the last fee operation (1072ms)
- ✓ adjustTrove(): borrower can't grief the baseRate and stop it decaying by issuing debt at higher frequency than the decay granularity (975ms)
- ✓ adjustTrove(): borrowing at non-zero base records the (drawn debt + fee) on the Trove struct (1070ms)
- ✓ adjustTrove(): Borrowing at non-zero base rate sends requested amount to the user (1092ms)
- ✓ adjustTrove(): Borrowing at zero base rate sends total requested USDE to the user (996ms)
- ✓ adjustTrove(): reverts when calling address has no active trove (622ms)
- ✓ adjustTrove(): reverts in Recovery Mode when the adjustment would reduce the TCR (723ms)
- ✓ adjustTrove(): collateral withdrawal reverts in Recovery Mode (454ms)
- ✓ adjustTrove(): debt increase that would leave ICR < 150% reverts in Recovery Mode (533ms)
- ✓ adjustTrove(): debt increase that would reduce the ICR reverts in Recovery Mode (673ms)
- ✓ adjustTrove(): A trove with ICR < CCR in Recovery Mode can adjust their trove to ICR > CCR (726ms)
- ✓ adjustTrove(): A trove with ICR > CCR in Recovery Mode can improve their ICR (638ms)
- ✓ adjustTrove(): debt increase in Recovery Mode charges 0.25% fee (645ms)

- ✓ adjustTrove(): reverts when change would cause the TCR of the system to fall below the CCR (456ms)
- ✓ adjustTrove(): reverts when USDE repaid is > debt of the trove (470ms)
- ✓ adjustTrove(): reverts when attempted ETH withdrawal is >= the trove's collateral (650ms)
- ✓ adjustTrove(): reverts when change would cause the ICR of the trove to fall below the MCR (559ms)
- ✓ adjustTrove(): With 0 coll change, doesnt change borrower's coll or ActivePool coll (369ms)
- ✓ adjustTrove(): With 0 debt change, doesnt change borrower's debt or ActivePool debt (362ms)
- ✓ adjustTrove(): updates borrower's debt and coll with an increase in both (586ms)
- ✓ adjustTrove(): updates borrower's debt and coll with a decrease in both (614ms)
- ✓ adjustTrove(): updates borrower's debt and coll with coll increase, debt decrease (570ms)
- ✓ adjustTrove(): updates borrower's debt and coll with coll decrease, debt increase (591ms)
- ✓ adjustTrove(): updates borrower's stake and totalStakes with a coll increase (638ms)
- ✓ adjustTrove(): updates borrower's stake and totalStakes with a coll decrease (500ms)
- ✓ adjustTrove(): changes USDEToken balance by the requested decrease (485ms)
- ✓ adjustTrove(): changes USDEToken balance by the requested increase (645ms)
- ✓ adjustTrove(): Changes the activePool ETH and raw ether balance by the requested decrease (508ms)
- ✓ adjustTrove(): Changes the activePool ETH and raw ether balance by the amount of ETH sent (514ms)
- ✓ adjustTrove(): Changes the USDE debt in ActivePool by requested decrease (767ms)
- ✓ adjustTrove(): Changes the USDE debt in ActivePool by requested increase (556ms)
- ✓ adjustTrove(): new coll = 0 and new debt = 0 is not allowed, as gas compensation still counts toward ICR (475ms)
- ✓ adjustTrove(): Reverts if requested debt increase and amount is zero (463ms)
- ✓ adjustTrove(): Reverts if requested coll withdrawal and ether is sent (355ms)
- ✓ adjustTrove(): Reverts if it's zero adjustment (203ms)
- ✓ adjustTrove(): Reverts if requested coll withdrawal is greater than trove's collateral (444ms)
- ✓ adjustTrove(): Reverts if borrower has insufficient USDE balance to cover his debt repayment (541ms)
- ✓ closeTrove(): reverts when it would lower the TCR below CCR (434ms)
- ✓ closeTrove(): reverts when calling address does not have active trove (350ms)
- ✓ closeTrove(): reverts when system is in Recovery Mode (754ms)
- ✓ closeTrove(): reverts when trove is the only one in the system (354ms)
- ✓ closeTrove(): reduces a Trove's collateral to zero (692ms)
- ✓ closeTrove(): reduces a Trove's debt to zero (654ms)
- ✓ closeTrove(): sets Trove's stake to zero (626ms)
- ✓ closeTrove(): zero's the troves reward snapshots (1380ms)
- ✓ closeTrove(): sets trove's status to closed and removes it from sorted troves list (642ms)
- ✓ closeTrove(): reduces ActivePool ETH and raw ether by correct amount (558ms)
- ✓ closeTrove(): reduces ActivePool debt by correct amount (566ms)
- ✓ closeTrove(): updates the the total stakes (804ms)
- ✓ closeTrove(): sends the correct amount of ETH to the user (552ms)
- ✓ closeTrove(): subtracts the debt of the closed Trove from the Borrower's USDEToken balance (637ms)
- ✓ closeTrove(): applies pending rewards (1606ms)
- ✓ closeTrove(): reverts if borrower has insufficient USDE balance to repay his entire debt (419ms)
- ✓ openTrove(): emits a TroveUpdated event with the correct collateral and debt (1066ms)
- ✓ openTrove(): Opens a trove with net debt >= minimum net debt (483ms)
- ✓ openTrove(): reverts if net debt < minimum net debt (200ms)

- ✓ openTrove(): decays a non-zero base rate (1024ms)
- ✓ openTrove(): doesn't change base rate if it is already zero (1128ms)
- ✓ openTrove(): lastFeeOpTime doesn't update if less time than decay interval has passed since the last fee operation (1040ms)
- ✓ openTrove(): reverts if max fee > 100% (80ms)
- ✓ openTrove(): reverts if max fee < 0.25% in Normal mode (116ms)
- ✓ openTrove(): reverts if fee exceeds max fee percentage (679ms)
- ✓ openTrove(): succeeds when fee is less than max fee percentage (1234ms)
- ✓ openTrove(): borrower can't grief the baseRate and stop it decaying by issuing debt at higher frequency than the decay granularity (1105ms)
- ✓ openTrove(): borrowing at non-zero base rate sends USDE fee to treasury contract (858ms)
- ✓ openTrove(): borrowing at non-zero base records the (drawn debt + fee + liq. reserve) on the Trove struct (982ms)
- ✓ openTrove(): Borrowing at non-zero base rate sends requested amount to the user (877ms)
- ✓ openTrove(): Borrowing at zero base rate charges minimum fee (488ms)
- ✓ openTrove(): reverts when system is in Recovery Mode and ICR < CCR (538ms)
- ✓ openTrove(): reverts when trove ICR < MCR (540ms)
- ✓ openTrove(): reverts when opening the trove would cause the TCR of the system to fall below the CCR (304ms)
- ✓ openTrove(): reverts if trove is already active (602ms)
- ✓ openTrove(): Can open a trove with ICR >= CCR when system is in Recovery Mode (673ms)
- ✓ openTrove(): Reverts opening a trove with min debt when system is in Recovery Mode (460ms)
- ✓ openTrove(): creates a new Trove and assigns the correct collateral and debt amount (204ms)
- ✓ openTrove(): adds Trove owner to TroveOwners array (188ms)
- ✓ openTrove(): creates a stake and adds it to total stakes (212ms)
- ✓ openTrove(): inserts Trove to Sorted Troves list (191ms)
- ✓ openTrove(): Increases the activePool ETH and raw ether balance by correct amount (328ms)
- ✓ openTrove(): records up-to-date initial snapshots of E_ETH and E_USDEDebt (754ms)
- ✓ openTrove(): allows a user to open a Trove, then close it, then re-open it (840ms)
- ✓ openTrove(): increases the Trove's USDE debt by the correct amount (196ms)
- ✓ openTrove(): increases USDE debt in ActivePool by the debt of the trove (261ms)
- ✓ openTrove(): increases user USDEToken balance by correct amount (162ms)
- ✓ getCompositeDebt(): returns debt + gas comp
- getNewICRFromTroveChange() returns the correct ICR
 - ✓ collChange = [0], debtChange = 0
 - ✓ collChange = 0, debtChange is positive
 - ✓ collChange = 0, debtChange is negative
 - ✓ collChange is positive, debtChange is 0
 - ✓ collChange is negative, debtChange is 0
 - ✓ collChange is negative, debtChange is negative
 - ✓ collChange is positive, debtChange is positive
 - ✓ collChange is positive, debtChange is negative
 - ✓ collChange is negative, debtChange is positive
- getNewTCRFromTroveChange() returns the correct TCR
 - ✓ collChange = 0, debtChange = 0 (763ms)
 - ✓ collChange = 0, debtChange is positive (588ms)
 - ✓ collChange = 0, debtChange is negative (704ms)
 - ✓ collChange is positive, debtChange is 0 (567ms)
 - ✓ collChange is negative, debtChange is 0 (670ms)
 - ✓ collChange is negative, debtChange is negative (567ms)
 - ✓ collChange is positive, debtChange is positive (579ms)
 - ✓ collChange is positive, debtChange is negative (594ms)
 - ✓ collChange is negative, debtChange is positive (695ms)

Without proxy

openTrove() multi collateral

- ✓ Open a trove with multiple collateral types, check if amounts added are correct (591ms)
- ✓ Open various troves with wrong order collateral, check that they have it properly (1346ms)
- ✓ Open trove multi collat with less balance than you have fails (54ms)
- ✓ Open trove after collateral price changes fails (804ms)

adjustTrove() multi collateral

- ✓ Open a trove with multiple collateral types, then adjust (2588ms)
- ✓ Adjusting trove without doing anything reverts (486ms)
- ✓ Adjusting trove by amounts / tokens that do not line up reverts (416ms)
- ✓ Adjusting trove by removing and adding same type of collateral does not work (342ms)
- ✓ Adjusting or opening trove with duplicate collat does not work (683ms)
- ✓ Adjusting a trove with collateral after price drops calculates VC correctly (1727ms)

check VC, TCR, balances multi collateral

- ✓ Open two multi collateral trove, check if collateral is correct (942ms)
- ✓ Open two multi collateral trove, check raw balances of contracts are correct (1129ms)
- ✓ Open multi collateral trove, adjust prices individually, check if VC and TCR change accordingly. Ratio 1 tokens (695ms)
- ✓ Ratio < 1, Open multi collateral trove, adjust prices individually, check if VC and TCR change accordingly (683ms)
- ✓ Including low decimal, Open multi collateral trove, adjust prices individually, check if VC and TCR change accordingly (656ms)
- ✓ When price drops to 0, VC updates accordingly. (590ms)

Various Test multi collateral borrow ops

Finished adding tokens and opening all troves.

Adjusting troves randomly.

Adjusting prices and adjusting troves again.

- ✓ Try various operations with lots of accounts and tokens. (21806ms)

Various Test fees borrow ops

- ✓ Basic get fee flat open trove before and after (1717ms)
- ✓ Basic get fee sloped open trove before and after (1525ms)
- ✓ Basic get fee sloped open and adjust trove before and after (1463ms)
- ✓ Test fee cap passed in to open trove (777ms)
- ✓ During bootstrapping period, max fee is 1% (1762ms)
- ✓ Test fee decay without much change in time. (1566ms)

Contract: CollSurplusPool

- ✓ CollSurplusPool::getETH(): Returns the ETH balance of the CollSurplusPool after redemption (2104ms)
- ✓ CollSurplusPool: claimColl(): Reverts if caller is not Borrower Operations
- ✓ CollSurplusPool: claimColl(): Reverts if nothing to claim
- ✓ CollSurplusPool: accountSurplus: reverts if caller is not Trove Manager

Contract: newBorrowerOperations

Without proxy

- ✓ removeCollateral(), loop through valid collateral and delete
- ✓ removeCollateral(), try remove collateral which is active
- ✓ getOracle(), check oracles for each collateral
- ✓ setOracle(), loop through valid collateral and change oracle
- ✓ Check after paused of collateral that people cannot open trove with that collateral, but it still works inside the system with adjust. (636ms)
- ✓ Check after paused of collateral that people cannot open trove with that collateral, but it still works inside the system. (1932ms)

Contract: Deployment script – Sets correct contract addresses dependencies after deployment

- ✓ Sets the correct USDEToken address in TroveManager
- ✓ Sets the correct SortedTrove address in TroveManager
- ✓ Sets the correct BorrowerOperations address in TroveManager
- ✓ Sets the correct ActivePool address in TroveManager
- ✓ Sets the correct DefaultPool address in TroveManager
- ✓ Sets the correct StabilityPool address in TroveManager
- ✓ Sets the correct StabilityPool address in ActivePool
- ✓ Sets the correct DefaultPool address in ActivePool
- ✓ Sets the correct BorrowerOperations address in ActivePool
- ✓ Sets the correct TroveManager address in ActivePool
- ✓ Sets the correct ActivePool address in StabilityPool
- ✓ Sets the correct BorrowerOperations address in StabilityPool
- ✓ Sets the correct USDEToken address in StabilityPool
- ✓ Sets the correct TroveManager address in StabilityPool
- ✓ Sets the correct TroveManager address in DefaultPool
- ✓ Sets the correct ActivePool address in DefaultPool
- ✓ Sets the correct TroveManager address in SortedTrove
- ✓ Sets the correct BorrowerOperations address in SortedTrove
- ✓ Sets the correct TroveManager address in BorrowerOperations
- ✓ Sets the correct SortedTrove address in BorrowerOperations
- ✓ Sets the correct ActivePool address in BorrowerOperations
- ✓ Sets the correct DefaultPool address in BorrowerOperations
- ✓ Sets the correct treasury address in BorrowerOperations

Contract: ERDMath

- ✓ max works if $a > b$
- ✓ max works if $a = b$
- ✓ max works if $a < b$

Contract: ERDSafeMath128Tester

- ✓ add(): reverts if overflows
- ✓ sub(): reverts if underflows

Contract: Fee arithmetic tests

- ✓ decayBaseRateFromBorrowing(): returns the initial base rate for no time increase
- ✓ decayBaseRateFromBorrowing(): returns the initial base rate for less than one minute passed (55ms)
- ✓ decayBaseRateFromBorrowing(): returns correctly decayed base rate, for various durations. Initial baseRate = 0.01 (1067ms)
- ✓ decayBaseRateFromBorrowing(): returns correctly decayed base rate, for various durations. Initial baseRate = 0.1 (975ms)
- ✓ decayBaseRateFromBorrowing(): returns correctly decayed base rate, for various durations. Initial baseRate = 0.34539284 (1056ms)
- ✓ decayBaseRateFromBorrowing(): returns correctly decayed base rate, for various durations. Initial baseRate = 0.9976 (1013ms)
- Basic exponentiation
 - ✓ decPow(): for exponent = 0, returns 1, regardless of base
 - ✓ decPow(): for exponent = 1, returns base, regardless of base
 - ✓ decPow(): for base = 0, returns 0 for any exponent other than 0 (79ms)
 - ✓ decPow(): for base = 1, returns 1 for any exponent (80ms)
 - ✓ decPow(): for exponent = 2, returns the square of the base
 - ✓ decPow(): correct output for various bases and exponents (188ms)
 - ✓ decPow(): abs. error $< 1e-9$ for exponent = 7776000 (seconds in three months) (2647ms)
 - ✓ decPow(): abs. error $< 1e-9$ for exponent = 2592000 (seconds in one month) (2574ms)
 - ✓ decPow(): abs. error $< 1e-9$ for exponent = 43200 (minutes in one month) (1756ms)
 - ✓ decPow(): abs. error $< 1e-9$ for exponent = 525600 (minutes in one year) (1968ms)
 - ✓ decPow(): abs. error $< 1e-9$ for exponent = 2628000 (minutes in five years) (2343ms)
 - ✓ decPow(): abs. error $< 1e-9$ for exponent = minutes in ten years (2418ms)
 - ✓ decPow(): abs. error $< 1e-9$ for exponent = minutes in one hundred years (2672ms)

Contract: Gas compensation tests

- ✓ `_getCollGasCompensation()`: returns the 0.5% of collateral if it is < \$10 in value
- ✓ `_getCollGasCompensation()`: returns 0.5% of collateral when 0.5% of collateral < \$10 in value
- ✓ `getCollGasCompensation()`: returns 0.5% of collateral when 0.5% of collateral = \$10 in value
- ✓ `getCollGasCompensation()`: returns 0.5% of collateral when 0.5% of collateral = \$10 in value
- ✓ `_getCompositeDebt()`: returns (debt + 50) when collateral < \$10 in value
- ✓ `getCompositeDebt()`: returns (debt + 50) collateral = \$10 in value
- ✓ `getCompositeDebt()`: returns (debt + 50) when 0.5% of collateral > \$10 in value
- ✓ `getCurrentICR()`: Incorporates virtual debt, and returns the correct ICR for new troves (1684ms)
- ✓ Gas compensation from pool-offset liquidations. All collateral paid as compensation (1957ms)
- ✓ gas compensation from pool-offset liquidations: 0.5% collateral < \$10 in value. Compensates \$10 worth of collateral, liquidates the remainder (1621ms)
- ✓ gas compensation from pool-offset liquidations: 0.5% collateral > \$10 in value. Compensates 0.5% of collateral, liquidates the remainder (1959ms)
- ✓ Gas compensation from pool-offset liquidations. Liquidation event emits the correct gas compensation and total liquidated coll and debt (1601ms)
- ✓ gas compensation from pool-offset liquidations. Liquidation event emits the correct gas compensation and total liquidated coll and debt (1795ms)
- ✓ gas compensation from pool-offset liquidations: 0.5% collateral > \$10 in value. Liquidation event emits the correct gas compensation and total liquidated coll and debt (1796ms)
- ✓ `liquidateTrove()`: full offset. Compensates the correct amount, and liquidates the remainder (2033ms)
- ✓ `liquidateTrove()`: full redistribution. Compensates the correct amount, and liquidates the remainder (1730ms)
- ✓ `liquidateTrove()`: full offset. Liquidation event emits the correct gas compensation and total liquidated coll and debt (2058ms)
- ✓ `liquidateTrove()`: full redistribution. Liquidation event emits the correct gas compensation and total liquidated coll and debt (2089ms)
- ✓ Trove ordering: same collateral, decreasing debt. Price successively increases. Troves should maintain ordering by ICR (2631ms)
- ✓ Trove ordering: increasing collateral, constant debt. Price successively increases. Troves should maintain ordering by ICR (4763ms)
- ✓ Trove ordering: Constant raw collateral ratio (excluding virtual debt). Price successively increases. Troves should maintain ordering by ICR (3644ms)

Contract: HintHelpers

- ✓ `setup`: makes accounts with nominal ICRs increasing by 1% consecutively (261ms)
 - ✓ `getApproxHint()`: returns the address of a Trove within $\sqrt{\text{length}}$ positions of the correct insert position (4121ms)
- Time-consuming
- ✓ `getApproxHint()`: for 10 random CRs, returns the address of a Trove within $\sqrt{\text{length}}$ positions of the correct insert position (10314ms)
 - ✓ `getApproxHint()`: returns the head of the list if the CR is the max uint256 value (1025ms)
 - ✓ `getApproxHint()`: returns the tail of the list if the CR is lower than ICR of any Trove (1025ms)

Contract: StabilityPool

Stability Pool Mechanisms

- ✓ `MULTICOLLATERAL withdrawFromSP()`: partial retrieval – retrieves correct USDE amount and the entire ETH Gain, and updates deposit (2944ms)
- ✓ `open`, deposit into SP, liquidate, deposit into SP #2, liquidate (3593ms)
- ✓ `withdrawFromSP()`: doesn't impact any troves, including the caller's trove (2480ms)
- ✓ `provideToSP()`, new deposit: depositor does not receive any collateral gains

(1701ms)

- ✓ provideToSP(): doesn't impact other users' deposits or ETH gains (3886ms)
- ✓ withdrawFromSP(): caller can withdraw full deposit and ETH gain during Recovery Mode (2844ms)

Contract: StabilityPool

- ✓ getETH(): gets the recorded ETH balance
- ✓ getTotalUSDEDeposits(): gets the recorded USDE balance

Contract: ActivePool

- ✓ getETH(): gets the recorded ETH balance
- ✓ getUSDEDebt(): gets the recorded USDE balance
- ✓ increaseUSDE(): increases the recorded USDE balance by the correct amount
- ✓ decreaseUSDE(): decreases the recorded USDE balance by the correct amount
- ✓ sendETH(): decreases the recorded ETH balance by the correct amount

Contract: DefaultPool

- ✓ getETH(): gets the recorded USDE balance
- ✓ getUSDEDebt(): gets the recorded USDE balance
- ✓ increaseUSDE(): increases the recorded USDE balance by the correct amount
- ✓ decreaseUSDE(): decreases the recorded USDE balance by the correct amount

Contract: PriceFeed

- ✓ C1 Chainlink working: fetchPrice should return the correct price, taking into account the number of decimal digits on the aggregator (109ms)
- ✓ C1 Chainlink breaks, Teller working: fetchPrice should return the correct Teller price, taking into account Teller's 6-digit granularity (109ms)
- ✓ C1 chainlinkWorking: Chainlink broken by zero latest roundId, Teller working: switch to usingChainlinkTellerUntrusted (45ms)
- ✓ C1 chainlinkWorking: Chainlink broken by zero latest roundId, Teller working: use Teller price (43ms)
- ✓ C1 chainlinkWorking: Chainlink broken by zero timestamp, Teller working, switch to usingChainlinkTellerUntrusted (42ms)
- ✓ C1 chainlinkWorking: Chainlink broken by zero timestamp, Teller working, return Teller price (39ms)
- ✓ C1 chainlinkWorking: Chainlink broken by future timestamp, Teller working, switch to usingChainlinkTellerUntrusted (42ms)
- ✓ C1 chainlinkWorking: Chainlink broken by future timestamp, Teller working, return Teller price (43ms)
- ✓ C1 chainlinkWorking: Chainlink broken by negative price, Teller working, switch to usingChainlinkTellerUntrusted
- ✓ C1 chainlinkWorking: Chainlink broken by negative price, Teller working, return Teller price
 - C1 chainlinkWorking: Chainlink broken - decimals call reverted, Teller working, switch to usingChainlinkTellerUntrusted
 - C1 chainlinkWorking: Chainlink broken - decimals call reverted, Teller working, return Teller price
 - C1 chainlinkWorking: Chainlink broken - latest round call reverted, Teller working, switch to usingChainlinkTellerUntrusted
 - C1 chainlinkWorking: latest round call reverted, Teller working, return the Teller price
- ✓ C1 chainlinkWorking: previous round call reverted, Teller working, switch to usingChainlinkTellerUntrusted (40ms)
- ✓ C1 chainlinkWorking: previous round call reverted, Teller working, return Teller Price (42ms)
- ✓ C1 chainlinkWorking: Chainlink frozen, Teller working: switch to usingTellerChainlinkFrozen (43ms)
- ✓ C1 chainlinkWorking: Chainlink frozen, Teller working: return Teller price (42ms)
- ✓ C1 chainlinkWorking: Chainlink frozen, Teller frozen: switch to usingTellerChainlinkFrozen (40ms)
- ✓ C1 chainlinkWorking: Chainlink frozen, Teller frozen: return last good price

- ✓ C1 chainlinkWorking: Chainlink times out, Tellor broken by 0 price: switch to usingChainlinkTellorUntrusted (40ms)
- ✓ C1 chainlinkWorking: Chainlink times out, Tellor broken by 0 price: return last good price
- ✓ C1 chainlinkWorking: Chainlink is out of date by <3hrs: remain chainlinkWorking
- ✓ C1 chainlinkWorking: Chainlink is out of date by <3hrs: return Chainlink price (38ms)
- ✓ C1 chainlinkWorking: Chainlink price drop of >50%, switch to usingChainlinkTellorUntrusted (40ms)
- ✓ C1 chainlinkWorking: Chainlink price drop of >50%, return the Tellor price (41ms)
- ✓ C1 chainlinkWorking: Chainlink price drop of 50%, remain chainlinkWorking (40ms)
- ✓ C1 chainlinkWorking: Chainlink price drop of 50%, return the Chainlink price (39ms)
- ✓ C1 chainlinkWorking: Chainlink price drop of <50%, remain chainlinkWorking (40ms)
- ✓ C1 chainlinkWorking: Chainlink price drop of <50%, return Chainlink price (40ms)
- ✓ C1 chainlinkWorking: Chainlink price increase of >100%, switch to usingChainlinkTellorUntrusted (41ms)
- ✓ C1 chainlinkWorking: Chainlink price increase of >100%, return Tellor price (41ms)
- ✓ C1 chainlinkWorking: Chainlink price increase of 100%, remain chainlinkWorking (40ms)
- ✓ C1 chainlinkWorking: Chainlink price increase of 100%, return Chainlink price (44ms)
- ✓ C1 chainlinkWorking: Chainlink price increase of <100%, remain chainlinkWorking (38ms)
- ✓ C1 chainlinkWorking: Chainlink price increase of <100%, return Chainlink price (40ms)
- ✓ C1 chainlinkWorking: Chainlink price drop of >50% and Tellor price matches: remain chainlinkWorking (40ms)
- ✓ C1 chainlinkWorking: Chainlink price drop of >50% and Tellor price matches: return Chainlink price (41ms)
- ✓ C1 chainlinkWorking: Chainlink price drop of >50% and Tellor price within 5% of Chainlink: remain chainlinkWorking (40ms)
- ✓ C1 chainlinkWorking: Chainlink price drop of >50% and Tellor price within 5% of Chainlink: return Chainlink price (41ms)
- ✓ C1 chainlinkWorking: Chainlink price drop of >50% and Tellor live but not within 5% of Chainlink: switch to usingChainlinkTellorUntrusted (42ms)
- ✓ C1 chainlinkWorking: Chainlink price drop of >50% and Tellor live but not within 5% of Chainlink: return Tellor price (42ms)
- ✓ C1 chainlinkWorking: Chainlink price drop of >50% and Tellor frozen: switch to usingChainlinkTellorUntrusted (42ms)
- ✓ C1 chainlinkWorking: Chainlink price drop of >50% and Tellor frozen: return last good price (45ms)
- ✓ C1 chainlinkWorking: Chainlink price drop of >50% and Tellor is broken by 0 price: switch to bothOracleSuspect
- ✓ C1 chainlinkWorking: Chainlink price drop of >50% and Tellor is broken by 0 price: return last good price (41ms)
- ✓ C1 chainlinkWorking: Chainlink price drop of >50% and Tellor is broken by 0 timestamp: switch to bothOracleSuspect
- ✓ C1 chainlinkWorking: Chainlink price drop of >50% and Tellor is broken by 0 timestamp: return last good price (43ms)
- ✓ C1 chainlinkWorking: Chainlink price drop of >50% and Tellor is broken by future timestamp: Pricefeed switches to bothOracleSuspect
- ✓ C1 chainlinkWorking: Chainlink price drop of >50% and Tellor is broken by future timestamp: return last good price (43ms)
- ✓ C1 chainlinkWorking: Chainlink is working and Tellor is working – remain on chainlinkWorking (40ms)
- ✓ C1 chainlinkWorking: Chainlink is working and Tellor is working – return Chainlink price (40ms)
- ✓ C1 chainlinkWorking: Chainlink is working and Tellor freezes – remain on chainlinkWorking (48ms)
- ✓ C1 chainlinkWorking: Chainlink is working and Tellor freezes – return Chainlink price (46ms)
- ✓ C1 chainlinkWorking: Chainlink is working and Tellor breaks: switch to

usingChainlinkTellorUntrusted (40ms)

- ✓ C1 chainlinkWorking: Chainlink is working and Tellor breaks: return Chainlink price (40ms)
- ✓ C2 usingTellorChainlinkUntrusted: Tellor breaks by zero price: switch to bothOraclesSuspect (42ms)
- ✓ C2 usingTellorChainlinkUntrusted: Tellor breaks by zero price: return last good price (42ms)
- ✓ C2 usingTellorChainlinkUntrusted: Tellor breaks by call reverted: switch to bothOraclesSuspect (41ms)
- ✓ C2 usingTellorChainlinkUntrusted: Tellor breaks by call reverted: return last good price (39ms)
- ✓ C2 usingTellorChainlinkUntrusted: Tellor breaks by zero timestamp: switch to bothOraclesSuspect (38ms)
- ✓ C2 usingTellorChainlinkUntrusted: Tellor breaks by zero timestamp: return last good price (42ms)
- ✓ C2 usingTellorChainlinkUntrusted: Tellor freezes – remain usingChainlinkTellorUntrusted (50ms)
- ✓ C2 usingTellorChainlinkUntrusted: Tellor freezes – return last good price (47ms)
- ✓ C2 usingTellorChainlinkUntrusted: both Tellor and Chainlink are live and $\leq 5\%$ price difference – switch to chainlinkWorking
- ✓ C2 usingTellorChainlinkUntrusted: both Tellor and Chainlink are live and $\leq 5\%$ price difference – return Chainlink price
- ✓ C2 usingTellorChainlinkUntrusted: both Tellor and Chainlink are live and $> 5\%$ price difference – remain usingChainlinkTellorUntrusted
- ✓ C2 usingTellorChainlinkUntrusted: both Tellor and Chainlink are live and $> 5\%$ price difference – return Tellor price
- ✓ C3 bothOraclesUntrusted: both Tellor and Chainlink are live and $> 5\%$ price difference remain bothOraclesSuspect
- ✓ C3 bothOraclesUntrusted: both Tellor and Chainlink are live and $> 5\%$ price difference, return last good price
- ✓ C3 bothOraclesUntrusted: both Tellor and Chainlink are live and $\leq 5\%$ price difference, switch to chainlinkWorking
- ✓ C3 bothOraclesUntrusted: both Tellor and Chainlink are live and $\leq 5\%$ price difference, return Chainlink price
- ✓ C4 usingTellorChainlinkFrozen: when both Chainlink and Tellor break, switch to bothOraclesSuspect
- ✓ C4 usingTellorChainlinkFrozen: when both Chainlink and Tellor break, return last good price (38ms)
- ✓ C4 usingTellorChainlinkFrozen: when Chainlink breaks and Tellor freezes, switch to usingChainlinkTellorUntrusted (42ms)
- ✓ C4 usingTellorChainlinkFrozen: when Chainlink breaks and Tellor freezes, return last good price (43ms)
- ✓ C4 usingTellorChainlinkFrozen: when Chainlink breaks and Tellor live, switch to usingChainlinkTellorUntrusted (40ms)
- ✓ C4 usingTellorChainlinkFrozen: when Chainlink breaks and Tellor live, return Tellor price (40ms)
- ✓ C4 usingTellorChainlinkFrozen: when Chainlink is live and Tellor is live with $< 5\%$ price difference, switch back to chainlinkWorking (40ms)
- ✓ C4 usingTellorChainlinkFrozen: when Chainlink is live and Tellor is live with $< 5\%$ price difference, return Chainlink current price (42ms)
- ✓ C4 usingTellorChainlinkFrozen: when Chainlink is live and Tellor is live with $> 5\%$ price difference, switch back to usingChainlinkTellorUntrusted (41ms)
- ✓ C4 usingTellorChainlinkFrozen: when Chainlink is live and Tellor is live with $> 5\%$ price difference, return Chainlink current price (41ms)
- ✓ C4 usingTellorChainlinkFrozen: when Chainlink is live and Tellor is live with similar price, switch back to chainlinkWorking (41ms)
- ✓ C4 usingTellorChainlinkFrozen: when Chainlink is live and Tellor is live with similar price, return Chainlink current price (42ms)
- ✓ C4 usingTellorChainlinkFrozen: when Chainlink is live and Tellor breaks, switch to usingChainlinkTellorUntrusted (39ms)
- ✓ C4 usingTellorChainlinkFrozen: when Chainlink is live and Tellor breaks, return

Chainlink current price (41ms)

- ✓ C4 usingTellerChainlinkFrozen: when Chainlink still frozen and Teller breaks, switch to usingChainlinkTellerUntrusted (46ms)

- ✓ C4 usingTellerChainlinkFrozen: when Chainlink still frozen and Teller broken, return last good price (47ms)

- ✓ C4 usingTellerChainlinkFrozen: when Chainlink still frozen and Teller live, remain usingTellerChainlinkFrozen (47ms)

- ✓ C4 usingTellerChainlinkFrozen: when Chainlink still frozen and Teller live, return Teller price (45ms)

- ✓ C4 usingTellerChainlinkFrozen: when Chainlink still frozen and Teller freezes, remain usingTellerChainlinkFrozen (45ms)

- ✓ C4 usingTellerChainlinkFrozen: when Chainlink still frozen and Teller freezes, return last good price (44ms)

- ✓ C5 usingChainlinkTellerUntrusted: when Chainlink is live and Teller price >5% - no status change (40ms)

- ✓ C5 usingChainlinkTellerUntrusted: when Chainlink is live and Teller price >5% - return Chainlink price (42ms)

- ✓ C5 usingChainlinkTellerUntrusted: when Chainlink is live and Teller price within <5%, switch to chainlinkWorking (41ms)

- ✓ C5 usingChainlinkTellerUntrusted: when Chainlink is live, Teller price not within 5%, return Chainlink price (40ms)

- ✓ C5 usingChainlinkTellerUntrusted: when Chainlink is live, <50% price deviation from previous, Teller price not within 5%, remain on usingChainlinkTellerUntrusted (41ms)

- ✓ C5 usingChainlinkTellerUntrusted: when Chainlink is live, <50% price deviation from previous, Teller price not within 5%, return Chainlink price (41ms)

- ✓ C5 usingChainlinkTellerUntrusted: when Chainlink is live, >50% price deviation from previous, Teller price not within 5%, remain on usingChainlinkTellerUntrusted (40ms)

- ✓ C5 usingChainlinkTellerUntrusted: when Chainlink is live, >50% price deviation from previous, Teller price not within 5%, return Chainlink price (46ms)

- ✓ C5 usingChainlinkTellerUntrusted: when Chainlink is live, <50% price deviation from previous, and Teller is frozen, remain on usingChainlinkTellerUntrusted (50ms)

- ✓ C5 usingChainlinkTellerUntrusted: when Chainlink is live, <50% price deviation from previous, Teller is frozen, return Chainlink price (50ms)

- ✓ C5 usingChainlinkTellerUntrusted: when Chainlink is live, >50% price deviation from previous, Teller is frozen, remain on usingChainlinkTellerUntrusted (49ms)

- ✓ C5 usingChainlinkTellerUntrusted: when Chainlink is live, >50% price deviation from previous, Teller is frozen, return Chainlink price (52ms)

- ✓ C5 usingChainlinkTellerUntrusted: when Chainlink frozen, remain on usingChainlinkTellerUntrusted (40ms)

- ✓ C5 usingChainlinkTellerUntrusted: when Chainlink frozen, return last good price

- ✓ C5 usingChainlinkTellerUntrusted: when Chainlink breaks too, switch to bothOraclesSuspect (44ms)

- ✓ C5 usingChainlinkTellerUntrusted: Chainlink breaks too, return last good price (38ms)

Mainnet PriceFeed setup

- ✓ fetchPrice should fail on contract with no chainlink address set

- ✓ fetchPrice should fail on contract with no tellor address set

- ✓ setAddresses should fail whe called by nonOwner

Contract: SortedTrove

SortedTrove

- ✓ contains(): returns true for addresses that have opened troves (541ms)

- ✓ contains(): returns false for addresses that have not opened troves (534ms)

- ✓ contains(): returns false for addresses that opened and then closed a trove (1142ms)

- ✓ contains(): returns true for addresses that opened, closed and then re-opened a trove (1669ms)

- ✓ contains(): returns false when there are no troves in the system

- ✓ contains(): true when list size is 1 and the trove the only one in system (198ms)

- ✓ contains(): false when list size is 1 and trove is not in the system (195ms)

BigNumber.toString does not accept any parameters; base-10 is assumed

- ✓ getMaxSize(): Returns the maximum list size
 - stays ordered after troves with 'infinite' ICR receive a redistribution

SortedTrove with mock dependencies

when params are wrongly set

- ✓ setParams(): reverts if size is zero

when params are properly set

- ✓ insert(): fails if list is full
- ✓ insert(): fails if list already contains the node
- ✓ insert(): fails if id is zero
- ✓ insert(): fails if ICR is zero
- ✓ remove(): fails if id is not in the list
- ✓ reInsert(): fails if list doesn't contain the node
- ✓ reInsert(): fails if new ICR is zero
- ✓ findInsertPosition(): No prevId for hint - ascend list starting from nextId, result is after the tail

Check position, re-insert multi-collateral multi-ratio, selective update, etc.

- ✓ findInsertPosition(): After price changes, list remains sorted as original. Update single trove updates just that ICR. (554ms)
- ✓ TroveManager updateTrove(), correctly inserts multiple troves. (668ms)

Contract: StabilityPool

Stability Pool Mechanisms

- ✓ provideToSP(): increases the Stability Pool USDE balance (222ms)
- ✓ provideToSP(): updates the user's deposit record in StabilityPool (228ms)
- ✓ provideToSP(): reduces the user's USDE balance by the correct amount (218ms)
- ✓ provideToSP(): increases totalUSDEDeposits by correct amount (204ms)
- ✓ provideToSP(): Correctly updates user snapshots of accumulated rewards per unit staked (1462ms)
- ✓ provideToSP(), multiple deposits: updates user's deposit and snapshots (1919ms)
- ✓ provideToSP(): reverts if user tries to provide more than their USDE balance (582ms)
- ✓ provideToSP(): reverts if user tries to provide $2^{256}-1$ USDE, which exceeds their balance (584ms)
- ✓ provideToSP(): reverts if cannot receive ETH Gain (1133ms)
- ✓ provideToSP(): doesn't impact other users' deposits or ETH gains (2043ms)
- ✓ provideToSP(): doesn't impact system debt, collateral or TCR (2114ms)
- ✓ provideToSP(): doesn't impact any troves, including the caller's trove (1651ms)
- ✓ provideToSP(): doesn't protect the depositor's trove from liquidation (1211ms)
- ✓ provideToSP(): providing 0 USDE reverts (882ms)
- ✓ provideToSP(), new deposit: when $SP > 0$, triggers Gain reward event - increases the sum G (795ms)
- ✓ provideToSP(), new deposit: when SP is empty, doesn't update G (834ms)
- ✓ provideToSP(), new deposit: sets the correct front end tag (1008ms)
- ✓ provideToSP(), new eligible deposit: tagged front end's stake increases (988ms)
- ✓ provideToSP(), new deposit: depositor does not receive ETH gains (886ms)
- ✓ provideToSP(), new deposit after past full withdrawal: depositor does not receive ETH gains (1629ms)
- ✓ provideToSP(), topup: triggers reward event - increases the sum G (802ms)
- ✓ provideToSP(), topup from different front end: doesn't change the front end tag (1049ms)
- ✓ provideToSP(), topup: tagged front end's stake increases (1652ms)
- ✓ provideToSP(): reverts when amount is zero (592ms)
- ✓ provideToSP(): reverts if user is a registered front end (756ms)
- ✓ provideToSP(): reverts if provided tag is not a registered front end (546ms)
- ✓ withdrawFromSP(): reverts when user has no active deposit (497ms)
- ✓ withdrawFromSP(): reverts when amount > 0 and system has an undercollateralized trove (439ms)
- ✓ withdrawFromSP(): partial retrieval - retrieves correct USDE amount and the entire ETH Gain, and updates deposit (1453ms)
- ✓ withdrawFromSP(): partial retrieval - leaves the correct amount of USDE in the Stability Pool (1422ms)

- ✓ withdrawFromSP(): full retrieval – leaves the correct amount of USDE in the Stability Pool (1268ms)
- ✓ withdrawFromSP(): Subsequent deposit and withdrawal attempt from same account, with no intermediate liquidations, withdraws zero ETH (1496ms)
- ✓ withdrawFromSP(): it correctly updates the user's USDE and ETH snapshots of entitled reward per unit staked (1453ms)
- ✓ withdrawFromSP(): decreases StabilityPool ETH (975ms)
- ✓ withdrawFromSP(): All depositors are able to withdraw from the SP to their account (2151ms)
- ✓ withdrawFromSP(): increases depositor's USDE token balance by the expected amount (2205ms)
- ✓ withdrawFromSP(): doesn't impact other users Stability deposits or ETH gains (1716ms)
- ✓ withdrawFromSP(): doesn't impact system debt, collateral or TCR (1741ms)
- ✓ withdrawFromSP(): doesn't impact any troves, including the caller's trove (1152ms)
- ✓ withdrawFromSP(): succeeds when amount is 0 and system has an undercollateralized trove (1260ms)
- ✓ withdrawFromSP(): withdrawing 0 USDE doesn't alter the caller's deposit or the total USDE in the Stability Pool (1061ms)
- ✓ withdrawFromSP(): withdrawing 0 ETH Gain does not alter the caller's ETH balance, their trove collateral, or the ETH in the Stability Pool (1391ms)
- ✓ withdrawFromSP(): Request to withdraw > caller's deposit only withdraws the caller's compounded deposit (1288ms)
- ✓ withdrawFromSP(): Request to withdraw $2^{256}-1$ USDE only withdraws the caller's compounded deposit (1530ms)
- ✓ withdrawFromSP(): caller can withdraw full deposit and ETH gain during Recovery Mode (1367ms)
- ✓ getDepositorETHGain(): depositor does not earn further ETH gains from liquidations while their compounded deposit == 0: (2358ms)
- ✓ withdrawFromSP(), partial withdrawal: doesn't change the front end tag (1335ms)
- ✓ withdrawFromSP(), partial withdrawal: tagged front end's stake decreases (1709ms)
- ✓ withdrawFromSP(), full withdrawal: removes deposit's front end tag (841ms)
- ✓ withdrawFromSP(), full withdrawal: zero's depositor's snapshots (1512ms)
- ✓ withdrawFromSP(), reverts when initial deposit value is 0 (978ms)
- ✓ registerFrontEnd(): registers the front end and chosen kickback rate (80ms)
- ✓ registerFrontEnd(): reverts if the front end is already registered
- ✓ registerFrontEnd(): reverts if the kickback rate >1
- ✓ registerFrontEnd(): reverts if address has a non-zero deposit already (572ms)

Contract: Pool Manager: Sum-Product rounding errors

- ✓ Rounding errors: 100 deposits of 100USDE into SP, then 200 liquidations of 49USDE (10277ms)

Contract: StabilityPool – Withdrawal of stability deposit – Reward calculations
Stability Pool Withdrawal

- ✓ withdrawFromSP(): Depositors with equal initial deposit withdraw correct compounded deposit and ETH Gain after one liquidation (821ms)
- ✓ withdrawFromSP(): Depositors with equal initial deposit withdraw correct compounded deposit and ETH Gain after two identical liquidations (1370ms)
- ✓ withdrawFromSP(): Depositors with equal initial deposit withdraw correct compounded deposit and ETH Gain after three identical liquidations (1695ms)
- ✓ withdrawFromSP(): Depositors with equal initial deposit withdraw correct compounded deposit and ETH Gain after two liquidations of increasing USDE (1345ms)
- ✓ withdrawFromSP(): Depositors with equal initial deposit withdraw correct compounded deposit and ETH Gain after three liquidations of increasing USDE (1665ms)
- ✓ withdrawFromSP(): Depositors with varying deposits withdraw correct compounded deposit and ETH Gain after two identical liquidations (1150ms)
- ✓ withdrawFromSP(): Depositors with varying deposits withdraw correct compounded deposit and ETH Gain after three identical liquidations (1528ms)
- ✓ withdrawFromSP(): Depositors with varying deposits withdraw correct compounded deposit and ETH Gain after three varying liquidations (1562ms)

- ✓ withdrawFromSP(): A, B, C Deposit -> 2 liquidations -> D deposits -> 1 liquidation. All deposits and liquidations = 100 USDE. A, B, C, D withdraw correct USDE deposit and ETH Gain (1682ms)
- ✓ withdrawFromSP(): A, B, C Deposit -> 2 liquidations -> D deposits -> 2 liquidations. All deposits and liquidations = 100 USDE. A, B, C, D withdraw correct USDE deposit and ETH Gain (2108ms)
- ✓ withdrawFromSP(): A, B, C Deposit -> 2 liquidations -> D deposits -> 2 liquidations. Various deposit and liquidation vals. A, B, C, D withdraw correct USDE deposit and ETH Gain (2104ms)
- ✓ withdrawFromSP(): A, B, C, D deposit -> 2 liquidations -> D withdraws -> 2 liquidations. All deposits and liquidations = 100 USDE. A, B, C, D withdraw correct USDE deposit and ETH Gain (1952ms)
- ✓ withdrawFromSP(): A, B, C, D deposit -> 2 liquidations -> D withdraws -> 2 liquidations. Various deposit and liquidation vals. A, B, C, D withdraw correct USDE deposit and ETH Gain (2156ms)
- ✓ withdrawFromSP(): A, B, D deposit -> 2 liquidations -> C makes deposit -> 1 liquidation -> D withdraws -> 1 liquidation. All deposits: 100 USDE. Liquidations: 100,100,100,50. A, B, C, D withdraw correct USDE deposit and ETH Gain (2187ms)
- ✓ withdrawFromSP(): Depositor withdraws correct compounded deposit after liquidation empties the pool (1258ms)
- ✓ withdrawFromSP(): Pool-emptying liquidation increases epoch by one, resets scaleFactor to 0, and resets P to 1e18 (1785ms)
- ✓ withdrawFromSP(): Depositors withdraw correct compounded deposit after liquidation empties the pool (1350ms)
- ✓ withdrawFromSP(): single deposit fully offset. After subsequent liquidations, depositor withdraws 0 deposit and *only* the ETH Gain from one liquidation (1344ms)
- ✓ withdrawFromSP(): Depositor withdraws correct compounded deposit after liquidation empties the pool (2779ms)
- ✓ withdrawFromSP(): deposit spans one scale factor change: Single depositor withdraws correct compounded deposit and ETH Gain after one liquidation (1281ms)
- ✓ withdrawFromSP(): Several deposits of varying amounts span one scale factor change. Depositors withdraw correct compounded deposit and ETH Gain after one liquidation (1671ms)
- ✓ withdrawFromSP(): deposit spans one scale factor change: Single depositor withdraws correct compounded deposit and ETH Gain after one liquidation (1151ms)
- ✓ withdrawFromSP(): Several deposits of varying amounts span one scale factor change. Depositors withdraws correct compounded deposit and ETH Gain after one liquidation (1458ms)
- alice deposit: 0
- ✓ withdrawFromSP(): Deposit that decreases to less than 1e-9 of it's original value is reduced to 0 (768ms)
- ✓ withdrawFromSP(): Several deposits of 10000 USDE span one scale factor change. Depositors withdraws correct compounded deposit and ETH Gain after one liquidation (2380ms)
- ✓ withdrawFromSP(): 2 depositors can withdraw after each receiving half of a pool-emptying liquidation (1940ms)
- ✓ withdrawFromSP(): Depositor's ETH gain stops increasing after two scale changes (2285ms)
- ✓ withdrawFromSP(): Large liquidated coll/debt, deposits and ETH price (987ms)
- ✓ withdrawFromSP(): Small liquidated coll/debt, large deposits and ETH price (1215ms)

Contract: StabilityPool – Withdrawal of stability deposit – Reward calculations Stability Pool Withdrawal

- ✓ withdrawCollateralGainToTrove(): Depositors with equal initial deposit withdraw correct compounded deposit and ETH Gain after one liquidation (1681ms)
- ✓ withdrawCollateralGainToTrove(): Depositors with equal initial deposit withdraw correct compounded deposit and ETH Gain after two identical liquidations (2142ms)
- ✓ withdrawCollateralGainToTrove(): Depositors with equal initial deposit withdraw correct compounded deposit and ETH Gain after three identical liquidations (2486ms)

- ✓ withdrawCollateralGainToTrove(): Depositors with equal initial deposit withdraw correct compounded deposit and ETH Gain after two liquidations of increasing USDE (1946ms)
- ✓ withdrawCollateralGainToTrove(): Depositors with equal initial deposit withdraw correct compounded deposit and ETH Gain after three liquidations of increasing USDE (2606ms)
- ✓ withdrawCollateralGainToTrove(): Depositors with varying deposits withdraw correct compounded deposit and ETH Gain after two identical liquidations (2017ms)
- ✓ withdrawCollateralGainToTrove(): Depositors with varying deposits withdraw correct compounded deposit and ETH Gain after three identical liquidations (2331ms)
- ✓ withdrawCollateralGainToTrove(): Depositors with varying deposits withdraw correct compounded deposit and ETH Gain after three varying liquidations (2332ms)
- ✓ withdrawCollateralGainToTrove(): A, B, C Deposit -> 2 liquidations -> D deposits -> 1 liquidation. All deposits and liquidations = 100 USDE. A, B, C, D withdraw correct USDE deposit and ETH Gain (2926ms)
- ✓ withdrawCollateralGainToTrove(): A, B, C Deposit -> 2 liquidations -> D deposits -> 2 liquidations. All deposits and liquidations = 100 USDE. A, B, C, D withdraw correct USDE deposit and ETH Gain (3359ms)
- ✓ withdrawCollateralGainToTrove(): A, B, C Deposit -> 2 liquidations -> D deposits -> 2 liquidations. Various deposit and liquidation vals. A, B, C, D withdraw correct USDE deposit and ETH Gain (3221ms)
- ✓ withdrawCollateralGainToTrove(): A, B, C, D deposit -> 2 liquidations -> D withdraws -> 2 liquidations. All deposits and liquidations = 100 USDE. A, B, C, D withdraw correct USDE deposit and ETH Gain (3502ms)
- ✓ withdrawCollateralGainToTrove(): A, B, C, D deposit -> 2 liquidations -> D withdraws -> 2 liquidations. Various deposit and liquidation vals. A, B, C, D withdraw correct USDE deposit and ETH Gain (2924ms)
- ✓ withdrawCollateralGainToTrove(): A, B, D deposit -> 2 liquidations -> C makes deposit -> 1 liquidation -> D withdraws -> 1 liquidation. All deposits: 100 USDE. Liquidations: 100,100,100,50. A, B, C, D withdraw correct USDE deposit and ETH Gain (2899ms)
- ✓ withdrawCollateralGainToTrove(): Depositor withdraws correct compounded deposit after liquidation empties the pool (2945ms)
- ✓ withdrawCollateralGainToTrove(): Pool-emptying liquidation increases epoch by one, resets scaleFactor to 0, and resets P to 1e18 (3043ms)
- ✓ withdrawCollateralGainToTrove(): Depositors withdraw correct compounded deposit after liquidation empties the pool (3214ms)
- ✓ withdrawCollateralGainToTrove(): single deposit fully offset. After subsequent liquidations, depositor withdraws 0 deposit and *only* the ETH Gain from one liquidation (2490ms)
- ✓ withdrawCollateralGainToTrove(): Depositor withdraws correct compounded deposit after liquidation empties the pool (5635ms)
- ✓ withdrawCollateralGainToTrove(): deposit spans one scale factor change: Single depositor withdraws correct compounded deposit and ETH Gain after one liquidation (1929ms)
- ✓ withdrawCollateralGainToTrove(): Several deposits of varying amounts span one scale factor change. Depositors withdraw correct compounded deposit and ETH Gain after one liquidation (2484ms)
- ✓ withdrawCollateralGainToTrove(): deposit spans one scale factor change: Single depositor withdraws correct compounded deposit and ETH Gain after one liquidation (1892ms)
- ✓ withdrawCollateralGainToTrove(): Several deposits of varying amounts span one scale factor change. Depositors withdraws correct compounded deposit and ETH Gain after one liquidation (2405ms)
- ✓ withdrawCollateralGainToTrove(): Deposit that decreases to less than 1e-9 of it's original value is reduced to 0 (1493ms)
- ✓ withdrawCollateralGainToTrove(): Several deposits of 10000 USDE span one scale factor change. Depositors withdraws correct compounded deposit and ETH Gain after one liquidation (3602ms)
- ✓ withdrawCollateralGainToTrove(): 2 depositors can withdraw after each receiving half of a pool-emptying liquidation (4167ms)
- ✓ withdrawCollateralGainToTrove(): Large liquidated coll/debt, deposits and ETH price (1245ms)
- ✓ withdrawCollateralGainToTrove(): Small liquidated coll/debt, large deposits and

ETH price (1218ms)

Contract: TroveManager

- ✓ liquidate(): closes a Trove that has ICR < MCR (871ms)
- ✓ liquidate(): decreases ActivePool ETH and USDEDebt by correct amounts (601ms)
- ✓ liquidate(): increases DefaultPool ETH and USDE debt by correct amounts (618ms)
- ✓ liquidate(): removes the Trove's stake from the total stakes (631ms)
- ✓ liquidate(): Removes the correct trove from the TroveOwners array, and moves the last array element to the new empty slot (1543ms)
- ✓ liquidate(): updates the snapshots of total stakes and total collateral (634ms)
- ✓ liquidate(): updates the E_ETH and E_USDEDebt reward-per-unit-staked totals (1287ms)
- ✓ liquidate(): Liquidates undercollateralized trove if there are two troves in the system (727ms)
- ✓ liquidate(): reverts if trove is non-existent (585ms)
- ✓ liquidate(): reverts if trove has been closed (793ms)
- ✓ liquidate(): does nothing if trove has $\geq 110\%$ ICR (543ms)
- ✓ liquidate(): Given the same price and no other trove changes, complete Pool offsets restore the TCR to its value prior to the defaulters opening troves (2628ms)
- ✓ liquidate(): Pool offsets increase the TCR (2994ms)
- ✓ liquidate(): a pure redistribution reduces the TCR only as a result of compensation (2786ms)
- ✓ liquidate(): does not affect the SP deposit or ETH gain when called on an SP depositor's address that has no trove (846ms)
- ✓ liquidate(): does not liquidate a SP depositor's trove with ICR > 110%, and does not affect their SP deposit or ETH gain (988ms)
- ✓ liquidate(): liquidates a SP depositor's trove with ICR < 110%, and the liquidation correctly impacts their SP deposit and ETH gain (1326ms)
- ✓ liquidate(): does not alter the liquidated user's token balance (1460ms)
- ✓ liquidate(): liquidates based on entire/collateral debt (including pending rewards), not raw collateral/debt (2219ms)
- ✓ liquidate(): when SP > 0, triggers Gain reward event – increases the sum G (1226ms)
- ✓ liquidate(): when SP is empty, doesn't update G (1243ms)
- ✓ batchLiquidateTrove(): liquidates a Trove that a) was skipped in a previous liquidation and b) has pending rewards (2754ms)
- ✓ batchLiquidateTrove(): closes every Trove with ICR < MCR, when n > number of undercollateralized troves (2711ms)
- ✓ batchLiquidateTrove(): liquidates up to the requested number of undercollateralized troves (1825ms)
- ✓ batchLiquidateTrove(): does nothing if all troves have ICR > 110% (1503ms)
- ✓ batchLiquidateTrove(): liquidates based on entire/collateral debt (including pending rewards), not raw collateral/debt (2039ms)
- ✓ batchLiquidateTrove(): reverts if _troveArray empty (931ms)
- ✓ batchLiquidateTrove(): liquidates troves with ICR < MCR (2356ms)
- ✓ batchLiquidateTrove(): does not affect the liquidated user's token balances (1611ms)
- ✓ batchLiquidateTrove(): A liquidation sequence containing Pool offsets increases the TCR (2886ms)
- ✓ batchLiquidateTrove(): A liquidation sequence of pure redistributions decreases the TCR, due to gas compensation, but up to 0.5% (2981ms)
- ✓ batchLiquidateTrove(): Liquidating troves with SP deposits correctly impacts their SP deposit and ETH gain (1407ms)
- ✓ batchLiquidateTrove(): when SP > 0, triggers Gain reward event – increases the sum G (1659ms)
- ✓ batchLiquidateTrove(): when SP is empty, doesn't update G (1738ms)
- ✓ batchLiquidateTrove(): closes every trove with ICR < MCR in the given array (2024ms)
- ✓ batchLiquidateTrove(): does not liquidate troves that are not in the given array (1698ms)
- ✓ batchLiquidateTrove(): does not close troves with ICR \geq MCR in the given array (2167ms)
- ✓ batchLiquidateTrove(): reverts if array is empty (1125ms)

- ✓ batchLiquidateTrove(): skips if trove is non-existent (1712ms)
- ✓ batchLiquidateTrove(): skips if a trove has been closed (1848ms)
- ✓ batchLiquidateTrove: when SP > 0, triggers Gain reward event – increases the sum G (1467ms)
- ✓ batchLiquidateTrove(): when SP is empty, doesn't update G (1467ms)
- ✓ getRedemptionHints(): gets the address of the first Trove and the final ICR of the last Trove involved in a redemption (976ms)
- ✓ getRedemptionHints(): When only one trove is updated it keeps the old icr values correctly. (938ms)
- ✓ getRedemptionHints(): returns 0 as partialRedemptionHintICR when reaching _maxIterations (980ms)
- ✓ redeemCollateral(): with invalid first hint, zero address (1526ms)
- ✓ redeemCollateral(): with invalid first hint, non-existent trove (1488ms)
- ✓ redeemCollateral(): with invalid first hint, trove below MCR (1582ms)
- ✓ redeemCollateral(): doesn't perform partial redemption if resultant debt would be < minimum net debt (2636ms)
- ✓ redeemCollateral(): doesn't perform the final partial redemption in the sequence if the hint is out-of-date (1777ms)
- ✓ redeemCollateral(): doesn't touch Troves with ICR < 110% (873ms)
- ✓ redeemCollateral(): finds the last Trove with ICR == 110% even if there is more than one (1803ms)
- ✓ redeemCollateral(): reverts when TCR < MCR (2531ms)
- ✓ redeemCollateral(): reverts when argument _amount is 0 (1356ms)
- ✓ redeemCollateral(): reverts if max fee > 100% (5111ms)
- ✓ redeemCollateral(): reverts if max fee < 0.5% (5800ms)
- ✓ redeemCollateral(): reverts if fee exceeds max fee percentage (7504ms)
- ✓ redeemCollateral(): succeeds if fee is less than max fee percentage (5773ms)
- ✓ redeemCollateral(): doesn't affect the Stability Pool deposits or ETH gain of redeemed-from troves (4425ms)
- ✓ redeemCollateral(): caller can redeem their entire USDEToken balance (1230ms)
- ✓ redeemCollateral(): reverts when requested redemption amount exceeds caller's USDE token balance (5971ms)
- ✓ redeemCollateral(): value of issued ETH == face value of redeemed USDE (assuming 1 USDE has value of \$1) (1961ms)
- ✓ redeemCollateral(): reverts if there is zero outstanding system debt
- ✓ redeemCollateral(): reverts if caller's tries to redeem more than the outstanding system debt (737ms)
- ✓ redeemCollateral(): a redemption made when base rate is zero increases the base rate (2683ms)
- ✓ redeemCollateral(): a redemption made when base rate is non-zero increases the base rate, for negligible time passed (5209ms)
- ✓ redeemCollateral(): lastFeeOpTime doesn't update if less time than decay interval has passed since the last fee operation [@skip-on-coverage] (6138ms)
- ✓ redeemCollateral(): a redemption made at zero base rate send a non-zero ETHFee to Gain staking contract (2420ms)
- ✓ redeemCollateral(): a redemption made at zero base increases the ETH-fees (2327ms)
- ✓ redeemCollateral(): a redemption made at a non-zero base rate send a non-zero ETHFee to contract (4204ms)
- ✓ redeemCollateral(): a redemption made at a non-zero base rate increases ETH-per (4033ms)
- ✓ redeemCollateral(): a redemption sends the ETH remainder (ETHDrawn – ETHFee) to the redeemer (2366ms)
- ✓ redeemCollateral(): a full redemption (leaving trove with 0 debt), closes the trove (2977ms)
- ✓ redeemCollateral(): emits correct debt and coll values in each redeemed trove's TroveUpdated event (3389ms)
- ✓ redeemCollateral(): a redemption that closes a trove leaves the trove's ETH surplus (collateral – ETH drawn) available for the trove owner to claim (3198ms)
- ✓ redeemCollateral(): a redemption that closes a trove leaves the trove's ETH surplus (collateral – ETH drawn) available for the trove owner after re-opening trove (3801ms)
- ✓ redeemCollateral(): reverts if fee eats up all returned collateral (2881ms)

- ✓ `getPendingUSDEDebtReward()`: Returns 0 if there is no pending USDEDebt reward (830ms)
- ✓ `getPendingETHReward()`: Returns 0 if there is no pending Coll reward (822ms)
- ✓ `computeICR()`: Returns 0 if trove's coll is worth 0
- ✓ `computeICR()`: Returns $2^{256}-1$ for ETH:USD = 100, coll = 1 ETH, debt = 100 USDE
- ✓ `computeICR()`: returns correct ICR for ETH:USD = 100, coll = 200 ETH, debt = 30 USDE
- ✓ `computeICR()`: returns correct ICR for ETH:USD = 250, coll = 1350 ETH, debt = 127

USDE

- ✓ `computeICR()`: returns correct ICR for ETH:USD = 100, coll = 1 ETH, debt = 54321 USDE
- ✓ `computeICR()`: Returns $2^{256}-1$ if trove has non-zero coll and zero debt
- ✓ `checkRecoveryMode()`: Returns true when TCR < 130% (802ms)
- ✓ `checkRecoveryMode()`: Returns false when TCR == 130% (446ms)
- ✓ `checkRecoveryMode()`: Returns false when TCR > 130% (434ms)
- ✓ `checkRecoveryMode()`: Returns false when TCR == 0 (435ms)
- ✓ `getTroveStake()`: Returns stake (401ms)
- ✓ `getTroveColl()`: Returns coll (669ms)
- ✓ `getTroveDebt()`: Returns debt (401ms)
- ✓ `getTroveStatus()`: Returns status (537ms)
- ✓ `hasPendingRewards()`: Returns false if trove is not active
- ✓ Single Collateral low ratio-partial to SP partial redistribution (1417ms)

Contract: TroveManager – Redistribution reward calculations

- ✓ redistribution: A, B Open. B Liquidated. C, D Open. D Liquidated. Distributes correct rewards (1417ms)
- ✓ redistribution: A, B, C Open. C Liquidated. D, E, F Open. F Liquidated. Distributes correct rewards (2106ms)
- ✓ redistribution: Sequence of alternate opening/liquidation: final surviving trove has ETH from all previously liquidated troves (2713ms)
- ✓ redistribution: A,B,C,D,E open. Liq(A). B adds coll. Liq(C). B and D have correct coll and debt (2994ms)
- ✓ redistribution: A,B,C,D open. Liq(A). B adds coll. Liq(C). B and D have correct coll and debt (2772ms)
- ✓ redistribution: A,B,C Open. Liq(C). B adds coll. Liq(A). B acquires all coll and debt (1820ms)
- ✓ redistribution: A,B,C Open. Liq(C). B tops up coll. D Opens. Liq(D). Distributes correct rewards. (1732ms)
- ✓ redistribution: Trove with the majority stake tops up. A,B,C, D open. Liq(D). C tops up. E Enters, Liq(E). Distributes correct rewards (1861ms)
- ✓ redistribution: Trove with the majority stake tops up. A,B,C, D open. Liq(D). A, B, C top up. E Enters, Liq(E). Distributes correct rewards (2506ms)
- ✓ redistribution: A,B,C Open. Liq(C). B withdraws coll. Liq(A). B acquires all coll and debt (1833ms)
- ✓ redistribution: A,B,C Open. Liq(C). B withdraws coll. D Opens. Liq(D). Distributes correct rewards. (1811ms)
- ✓ redistribution: Trove with the majority stake withdraws. A,B,C,D open. Liq(D). C withdraws some coll. E Enters, Liq(E). Distributes correct rewards (1870ms)
- ✓ redistribution: Trove with the majority stake withdraws. A,B,C,D open. Liq(D). A, B, C withdraw. E Enters, Liq(E). Distributes correct rewards (2694ms)
- ✓ redistribution, all operations: A,B,C open. Liq(A). D opens. B adds, C withdraws. Liq(B). E & F open. D adds. Liq(F). Distributes correct rewards (3128ms)
- ✓ redistribution, all operations: A,B,C open. Liq(A). D opens. B adds, C withdraws. Liq(B). E & F open. D adds. Liq(F). Varying coll. Distributes correct rewards (3038ms)
- ✓ open alice and bob (1411ms)
- ✓ multi redistribution: A, B Open. B Liquidated. C, D Open. D Liquidated. Distributes correct rewards (1758ms)
- ✓ multi redistribution: A open with ETH, B open with STETH, C open with both, liquidate C (1135ms)
- ✓ multi redistribution `revert()`: A open with ETH, B open with STETH, C open with both, liquidate C, then liquidate B (1404ms)
- ✓ multi redistribution: A open with ETH, B open with STETH, C open with both 5:1.5 ratio, liquidate C (1327ms)

Contract: TroveManager – in Recovery Mode

- ✓ checkRecoveryMode(): Returns true if TCR falls below CCR (501ms)
- ✓ checkRecoveryMode(): Returns true if TCR stays less than CCR (665ms)
- ✓ checkRecoveryMode(): returns false if TCR stays above CCR (629ms)
- ✓ checkRecoveryMode(): returns false if TCR rises above CCR (677ms)
- ✓ liquidate(), with $ICR < 100\%$: removes stake and updates totalStakes (771ms)
- ✓ liquidate(), with $ICR < 100\%$: updates system snapshots correctly (1592ms)
- ✓ liquidate(), with $ICR < 100\%$: closes the Trove and removes it from the Trove array (805ms)
- ✓ liquidate(), with $ICR < 100\%$: only redistributes to active Troves – no offset to Stability Pool (894ms)
- ✓ liquidate(), with $100 < ICR < 110\%$: removes stake and updates totalStakes (775ms)
- ✓ liquidate(), with $100\% < ICR < 110\%$: updates system snapshots correctly (1227ms)
- ✓ liquidate(), with $100\% < ICR < 110\%$: closes the Trove and removes it from the Trove array (1071ms)
- ✓ liquidate(), with $100\% < ICR < 110\%$: offsets as much debt as possible with the Stability Pool, then redistributes the remainder coll and debt (1016ms)
- ✓ liquidate(), with $ICR > 110\%$, trove has lowest ICR, and StabilityPool is empty: does nothing (881ms)
- ✓ liquidate(), with $110\% < ICR < TCR$, and StabilityPool USDE > debt to liquidate: offsets the trove entirely with the pool (1130ms)
- ✓ liquidate(), with $ICR = 110 < TCR$, and StabilityPool USDE > debt to liquidate: offsets the trove entirely with the pool, there's no collateral surplus (1156ms)
- ✓ liquidate(), with $110\% < ICR < TCR$, and StabilityPool USDE > debt to liquidate: removes stake and updates totalStakes (1112ms)
- ✓ liquidate(), with $110\% < ICR < TCR$, and StabilityPool USDE > debt to liquidate: updates system snapshots (1266ms)
- ✓ liquidate(), with $110\% < ICR < TCR$, and StabilityPool USDE > debt to liquidate: closes the Trove (1078ms)
- ✓ liquidate(), with $110\% < ICR < TCR$, and StabilityPool USDE > debt to liquidate: can liquidate troves out of order (3031ms)
- ✓ liquidate(), with $ICR > 110\%$, and StabilityPool USDE < liquidated debt: Trove remains active (780ms)
- ✓ liquidate(), with $ICR > 110\%$, and StabilityPool USDE < liquidated debt: Trove remains in TroveOwners array (796ms)
- ✓ liquidate(), with $ICR > 110\%$, and StabilityPool USDE < liquidated debt: nothing happens (839ms)
- ✓ liquidate(), with $ICR > 110\%$, and StabilityPool USDE < liquidated debt: updates system snapshots (754ms)
- ✓ liquidate(), with $ICR > 110\%$, and StabilityPool USDE < liquidated debt: causes correct Pool offset and ETH gain, and doesn't redistribute to active troves (785ms)
- ✓ liquidate(), with $ICR > 110\%$, and StabilityPool USDE < liquidated debt: ICR of non liquidated trove does not change (1700ms)
- ✓ liquidate() with $ICR > 110\%$, and StabilityPool USDE < liquidated debt: total liquidated coll and debt is correct (1452ms)
- ✓ liquidate(): Doesn't liquidate undercollateralized trove if it is the only trove in the system (780ms)
- ✓ liquidate(): Liquidates undercollateralized trove if there are two troves in the system (832ms)
- ✓ liquidate(): does nothing if trove has $\geq 110\%$ ICR and the Stability Pool is empty (837ms)
- ✓ liquidate(): does nothing if trove $ICR \geq TCR$, and SP covers trove's debt (1235ms)
- ✓ liquidate(): reverts if trove is non-existent (445ms)
- ✓ liquidate(): reverts if trove has been closed (959ms)
- ✓ liquidate(): liquidates based on entire/collateral debt (including pending rewards), not raw collateral/debt (2077ms)
- ✓ liquidate(): does not affect the SP deposit or ETH gain when called on an SP depositor's address that has no trove (848ms)
- ✓ liquidate(): does not alter the liquidated user's token balance (1678ms)
- ✓ liquidate(), with $110\% < ICR < TCR$, can claim collateral, re-open, be redeemed and claim again (3327ms)

- ✓ liquidate(), with $110\% < \text{ICR} < \text{TCR}$, can claim collateral, after another claim from a redemption (3576ms)
- ✓ liquidateTrove(): With all ICRs $> 110\%$, Liquidates Troves until system leaves recovery mode (2781ms)
- ✓ liquidateTrove(): Liquidates Troves until 1) system has left recovery mode AND 2) it reaches a Trove with $\text{ICR} \geq 110\%$ (2427ms)
- ✓ liquidateTrove(): liquidates only up to the requested number of undercollateralized troves (1854ms)
- ✓ liquidateTrove(): does nothing if $n = 0$ (1003ms)
- ✓ liquidateTrove(): closes every Trove with $\text{ICR} < \text{MCR}$, when $n >$ number of undercollateralized troves (2643ms)
- ✓ liquidateTrove(): a liquidation sequence containing Pool offsets increases the TCR (3109ms)
- ✓ liquidateTrove(): A liquidation sequence of pure redistributions decreases the TCR, due to gas compensation, but up to 0.5% (2490ms)
- ✓ liquidateTrove(): liquidates based on entire/collateral debt (including pending rewards), not raw collateral/debt (1849ms)
- ✓ liquidateTrove(): does nothing if all troves have $\text{ICR} > 110\%$ and Stability Pool is empty (915ms)
- ✓ liquidateTrove(): emits liquidation event with correct values when all troves have $\text{ICR} > 110\%$ and Stability Pool covers a subset of troves (2358ms)
- ✓ liquidateTrove(): emits liquidation event with correct values when all troves have $\text{ICR} > 110\%$ and Stability Pool covers a subset of troves, including a partial (2889ms)
- ✓ liquidateTrove(): does not affect the liquidated user's token balances (1403ms)
- ✓ liquidateTrove(): Liquidating troves at $100 < \text{ICR} < 110$ with SP deposits correctly impacts their SP deposit and ETH gain (1849ms)
- ✓ liquidateTrove(): Liquidating troves at $\text{ICR} \leq 100\%$ with SP deposits does not alter their deposit or ETH gain (2288ms)
- ✓ liquidateTrove() with a non fullfilled liquidation: non liquidated trove remains active (2198ms)
- ✓ liquidateTrove() with a non fullfilled liquidation: non liquidated trove remains in TroveOwners Array (1775ms)
- ✓ liquidateTrove() with a non fullfilled liquidation: still can liquidate further troves after the non-liquidated, emptied pool (2436ms)
- ✓ liquidateTrove() with a non fullfilled liquidation: still can liquidate further troves after the non-liquidated, non emptied pool (2578ms)
- ✓ liquidateTrove() with a non fullfilled liquidation: total liquidated coll and debt is correct (2121ms)
- ✓ liquidateTrove() with a non fullfilled liquidation: emits correct liquidation event values (2023ms)
- ✓ liquidateTrove() with a non fullfilled liquidation: ICR of non liquidated trove does not change (2405ms)
- ✓ batchLiquidateTrove(): Liquidates all troves with $\text{ICR} < 110\%$, transitioning Normal -> Recovery Mode (2530ms)
- ✓ batchLiquidateTrove(): Liquidates all troves with $\text{ICR} < 110\%$, transitioning Recovery -> Normal Mode (2495ms)
- ✓ batchLiquidateTrove(): Liquidates all troves with $\text{ICR} < 110\%$, transitioning Normal -> Recovery Mode (2977ms)
- ✓ batchLiquidateTrove() with a non fullfilled liquidation: non liquidated trove remains active (2286ms)
- ✓ batchLiquidateTrove() with a non fullfilled liquidation: non liquidated trove remains in Trove Owners array (2206ms)
- ✓ batchLiquidateTrove() with a non fullfilled liquidation: still can liquidate further troves after the non-liquidated, emptied pool (2289ms)
- ✓ batchLiquidateTrove() with a non fullfilled liquidation: still can liquidate further troves after the non-liquidated, non emptied pool (2237ms)
- ✓ batchLiquidateTrove() with a non fullfilled liquidation: total liquidated coll and debt is correct (1934ms)
- ✓ batchLiquidateTrove() with a non fullfilled liquidation: emits correct liquidation event values (1868ms)
- ✓ batchLiquidateTrove() with a non fullfilled liquidation: ICR of non liquidated

trove does not change (2076ms)

- ✓ batchLiquidateTrove(), with $110\% < ICR < TCR$, and StabilityPool USDE > debt to liquidate: can liquidate troves out of order (2353ms)

- ✓ batchLiquidateTrove(), with $110\% < ICR < TCR$, and StabilityPool empty: doesn't liquidate any troves (1682ms)

- ✓ batchLiquidateTrove(): skips liquidation of troves with $ICR > TCR$, regardless of Stability Pool size (4456ms)

- ✓ batchLiquidateTrove(): emits liquidation event with correct values when all troves have $ICR > 110\%$ and Stability Pool covers a subset of troves (2642ms)

- ✓ batchLiquidateTrove(): emits liquidation event with correct values when all troves have $ICR > 110\%$ and Stability Pool covers a subset of troves, including a partial (2779ms)

Contract: TroveManager – in Recovery Mode – back to normal mode in 1 tx

Batch liquidations

- ✓ First trove only doesn't get out of Recovery Mode (1504ms)

- ✓ Two troves over MCR are liquidated (1524ms)

- ✓ Stability Pool profit matches (1606ms)

- ✓ A trove over TCR is not liquidated (1568ms)

Sequential liquidations

- ✓ First trove only doesn't get out of Recovery Mode (1073ms)

- ✓ Two troves over MCR are liquidated (1134ms)

Contract: USDEToken

Basic token functions, without Proxy

- ✓ balanceOf(): gets the balance of the account

- ✓ totalSupply(): gets the total supply

- ✓ name(): returns the token's name

- ✓ symbol(): returns the token's symbol

- ✓ decimal(): returns the number of decimal digits used

- ✓ allowance(): returns an account's spending allowance for another account's balance

- ✓ approve(): approves an account to spend the specified amount

- ✓ approve(): reverts when spender param is address(0)

- ✓ approve(): reverts when owner param is address(0)

- ✓ transferFrom(): successfully transfers from an account which is it approved to transfer from (39ms)

- ✓ transfer(): increases the recipient's balance by the correct amount

- ✓ transfer(): reverts if amount exceeds sender's balance

- ✓ transfer(): transferring to a blacklisted address reverts

- ✓ increaseAllowance(): increases an account's allowance by the correct amount

- ✓ mint(): issues correct amount of tokens to the given address

- ✓ burn(): burns correct amount of tokens from the given address

- ✓ sendToPool(): changes balances of Stability pool and user by the correct amounts

- ✓ returnFromPool(): changes balances of Stability pool and user by the correct amounts

amounts

- ✓ transfer(): transferring to a blacklisted address reverts

- ✓ decreaseAllowance(): decreases allowance by the expected amount

- ✓ decreaseAllowance(): fails trying to decrease more than previously allowed

- ✓ version(): returns the token contract's version

- ✓ Initializes PERMIT_TYPEHASH correctly

- ✓ Initializes DOMAIN_SEPARATOR correctly

- ✓ Initial nonce for a given address is 0

- ✓ permits and emits an Approval event (replay protected) (39ms)

- ✓ permits(): fails with expired deadline

- ✓ permits(): fails with the wrong signature

Contract: newBorrowerOperations

Without proxy

amounts[0] 2002500000000000000

amounts[1] 2002500000000000000

trove coll address 0xaA6C186296d76C1E6EA4Ae71ff0809E96f31b9Ed

trove coll amount 2002500000000000000

```
WETH active pool has: 20
steth activepool has: 20
usde MINTED: 3795
steth alice has: 0
weth alice has: 0
trove coll address 0xaA6C186296d76C1E6EA4Ae71ff0809E96f31b9Ed
trove coll amount 10200250000000000000000000
Trove debt2: 4005000002238314941654
trove coll address
0xaA6C186296d76C1E6EA4Ae71ff0809E96f31b9Ed,0x8Fd726bb2B049A1fbE8b456222A1496ce6973Ae3
trove coll amount 2020025000000000000000000,20025000000000000000
Trove debt3: 6010000011287496839724
5795
6795
result true
WETH active pool has: 20
steth activepool has: 20
usde MINTED: 985
steth alice has: 20
weth alice has: 2020
Trove debt: 0
    ✓ addColl(), basic sanity (1466ms)

Contract: BorrowerOperations
Without proxy
    ✓ addColl(): reverts when top-up would leave trove with ICR < MCR (567ms)

780 passing (1h)
5 pending
```

Code Coverage

Calculating code coverage was not possible with the current repository: the contracts require compilation with `--via-ir`.

Changelog

- 2023-10-05 - Initial report
- 2023-10-25 - Updated report
- 2023-10-30 - Final report

About Quantstamp

Quantstamp is a global leader in blockchain security. Founded in 2017, Quantstamp's mission is to securely onboard the next billion users to Web3 through its best-in-class Web3 security products and services.

Quantstamp's team consists of cybersecurity experts hailing from globally recognized organizations including Microsoft, AWS, BMW, Meta, and the Ethereum Foundation. Quantstamp engineers hold PhDs or advanced computer science degrees, with decades of combined experience in formal verification, static analysis, blockchain audits, penetration testing, and original leading-edge research.

To date, Quantstamp has performed more than 500 audits and secured over \$200 billion in digital asset risk from hackers. Quantstamp has worked with a diverse range of customers, including startups, category leaders and financial institutions. Brands that Quantstamp has worked with include Ethereum 2.0, Binance, Visa, PayPal, Polygon, Avalanche, Curve, Solana, Compound, Lido, MakerDAO, Arbitrum, OpenSea and the World Economic Forum.

Quantstamp's collaborations and partnerships showcase our commitment to world-class research, development and security. We're honored to work with some of the top names in the industry and proud to secure the future of web3.

Notable Collaborations & Customers:

- Blockchains: Ethereum 2.0, Near, Flow, Avalanche, Solana, Cardano, Binance Smart Chain, Hedera Hashgraph, Tezos
- DeFi: Curve, Compound, Maker, Lido, Polygon, Arbitrum, SushiSwap
- NFT: OpenSea, Parallel, Dapper Labs, Decentraland, Sandbox, Axie Infinity, Illuvium, NBA Top Shot, Zora
- Academic institutions: National University of Singapore, MIT

Timeliness of content

The content contained in the report is current as of the date appearing on the report and is subject to change without notice, unless indicated otherwise by Quantstamp; however, Quantstamp does not guarantee or warrant the accuracy, timeliness, or completeness of any report you access using the internet or other means, and assumes no obligation to update any information following publication or other making available of the report to you by Quantstamp.

Notice of confidentiality

This report, including the content, data, and underlying methodologies, are subject to the confidentiality and feedback provisions in your agreement with Quantstamp. These materials are not to be disclosed, extracted, copied, or distributed except to the extent expressly authorized by Quantstamp.

Links to other websites

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Quantstamp. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that Quantstamp are not responsible for the content or operation of such web sites, and that Quantstamp shall have no liability to you or any other person or entity for the use of third-party web sites. Except as described below, a hyperlink from this web site to another web site does not imply or mean that Quantstamp endorses the content on that web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the report. Quantstamp assumes no responsibility for the use of third-party software on any website and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any output generated by such software.

Disclaimer

The review and this report are provided on an as-is, where-is, and as-available basis. To the fullest extent permitted by law, Quantstamp disclaims all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. You agree that your access and/or use of the report and other results of the review, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE. This report is based on the scope of materials and documentation provided for a limited review at the time provided. You acknowledge that Blockchain technology remains under development and is subject to unknown risks and flaws and, as such, the report may not be complete or inclusive of all vulnerabilities. The review is limited to the materials identified in the report and does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. The report does not indicate the endorsement by Quantstamp of any particular project or team, nor guarantee its security, and may not be represented as such. No third party is entitled to rely on the report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. Quantstamp does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party, or any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, or any related services and products, any hyperlinked websites, or any other websites or mobile applications, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third party. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate.



Quantstamp

© 2023 – Quantstamp, Inc.

Ethereum Reserve Dollar (ERD)