Three Sigma Labs

Code Audit

Borrowing Protocol

Yeti Finance

Disclaimer

The ensuing audit offers no assertions or assurances about the code's security. It cannot be deemed an adequate judgment of the contract's correctness on its own. The authors of this audit present it solely as an informational exercise, reporting the thorough research involved in the secure development of the intended contracts, and make no material claims or guarantees regarding the contract's post-deployment operation. The authors of this report disclaim all liability for all kinds of potential consequences of the contract's deployment or use. Due to the possibility of human error occurring during the code’s manual review process, we advise the client team to commission several independent audits in addition to a public bug bounty program.

Table of Contents

Summary

Three Sigma Labs audited Yeti Finance's borrowing protocol smart contracts in a 5 person week engagement. The audit was conducted from 07-03-2022 to 20-03-2022.

Protocol Description

Yeti Finance allows users to borrow YUSD, a USD-pegged stablecoin by trustlessly supplying arbitrary whitelisted crypto-assets as collateral with interest-free liquidity and a low minimum collateralization ratio of 110%. Yeti Finance, similarly to Liquity, uses a pooled liquidation mechanism that shifts risk and capital requirements from liquidators to Stability Pool depositors, who stake YUSD and earn revenue from liquidations without having to interact with complex mechanisms such as collateral auctions. Along with Stability Pool direct liquidations, the protocol also incorporates a redemption mechanism that effectively creates a lower-bound on the price of YUSD. However, unlike Liquity, which is limited to accepting only ETH as collateral, Yeti Finance allows the user to supply a portfolio of a wide range of assets to be used as collateral. This new design implies several major implementation changes from Liquity's.

Scope

The audit examined only the Core contracts of Yeti Finance's borrowing protocol. Due to time constraints for the execution of the audit but the need for an holistic review of the protocol, the code was divided into separate priorities.

The following Core contracts were considered **high** priority:

- BorrowerOperations.sol
- SortedTroves.sol
- TroveManagerLiquidations.sol
- TroveManagerRedemptions.sol
- StabilityPool.sol
- YetiController.sol

The following contracts were considered **medium** priority:

- ActivePool.sol
- TroveManager.sol
- PriceFeed.sol
- ThreePieceWiseLinearFeeCurve.sol
- YetiCustomBase.sol

The following contracts were considered **low** priority:

- CollSurplusPool.sol
- DefaultPool.sol
- GasPool.sol
- YUSDToken.sol
- YETIToken.sol
- TeamAllocation.sol
- LinearVest.sol
- Timelock.sol

The review was conducted on the code present in a private repository shared with Three Sigma, which contains a Hardhat project with test scripts as well as a document providing additional information. The code was frozen for review at commit **65d6d9243ed4d2ccc1ae78d32e15766de5f9f1ea**.

Assumptions

The SortedTroves.sol contract contains a list of the system's Troves in the order determined by AICR. This list can be sorted by calling the Trove Manager contract's external method updateTroves. Keepers of this function, specially the Yeti Finance team, are responsible for calling this method on a regular basis to ensure that the list is sorted. While the Yeti Finance team cannot guarantee that the list is always sorted, we are assuming it is throughout the course of the audit.

Additionally, it was assumed that when troves become liquidatable, they are liquidated by an external actor in a timely manner.

To sum up, it was assumed that every mechanism outside the scope of this document operated flawlessly.

Methodology

To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Thirdly, we evaluated invariants that should apply to Yeti Finance and validated them using the Echidna fuzzing tool. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

In this audit we report our findings using as a guideline Immunefi's vulnerability taxonomy which can be found at immunefi.com/severity-updated/.

Level	Description
Critical	- Empty or freeze the contract's holdings. - Cryptographic flaws.
High	- Token holders temporarily unable to transfer holdings. - Users spoof each other. - Theft of yield. - Transient consensus failures.
Medium	- Contract consumes unbounded gas. - Block stuffing. - Griefing denial of service. - Gas griefing.
Low	- Contract fails to deliver promised returns, but doesn't lose value.
None	- Best practices.

Project Dashboard

Application Summary

Name	Yeti Finance
Commit	65d6d924
Language	Solidity
Platform	Avalanche

Engagement Summary

Timeline	7 March to 20 March, 2022
Nº of Auditors	2
Review Time	5 person weeks

Vulnerability Summary

Nº Critical Severity Issues	2
Nº Medium Severity Issues	0
Nº Low Severity Issues	4
Nº Informational Severity Issues	Several (Slither)

Category Breakdown

Functional Correctness	5
Access Controls	1
Best Practice	Several (Slither)

Code Maturity Evaluation

Category	Evaluation
Access Controls	**Satisfactory.** The codebase has a strong access control mechanism.
Arithmetic	**Satisfactory.** The codebase uses SafeMath to prevent overflows and underflows and presents insignificant losses in precision.
Centralization	**Weak.** The Yeti Finance team has significant privileges over the protocol.
Code Stability	**Moderate.** The code was altered during the audit.
Upgradeability	**Moderate.** Certain parameterizations of the system can be modified after deployment.
Function Composition	**Moderate.** Certain components are similar, and the codebase would benefit from increased code reuse.
Front-Running	**Moderate.** Transactions that use leverage can be profitable to frontrun due to token swapping on DEX's. Borrowers can also frontrun redeem transactions to avoid being redeemed against.
Monitoring	**Satisfactory.** Events are correctly emitted.
Specification	**Weak.** Numerous behaviors were excluded from the available documentation, and the codebase will further benefit from more thorough documentation.
Testing and Verification	**Moderate.** Numerous tests were failing, missing, or outdated. These tests were either fixed or created by the team during the audit.

Automated Testing and Verification

To enhance coverage of certain areas of the codebase we complement our analysis with a set of automated testing techniques:

- **Slither:** A Solidity static analysis framework with native support for multiple vulnerability detectors. We used Slither to scan the entire codebase against common vulnerabilities and programming malpractices.

- **Echidna:** A fuzzer for Ethereum smart contracts, capable of generating grammar-based fuzzing campaigns based on contract ABI to falsify user-defined properties and Solidity assertions. We used Echidna to test the expected properties of the core components of the Yeti borrowing protocol, including borrower operations, trove manager and stability pool.

Despite augmenting our security analysis, automated testing techniques still present some limitations and should therefore not be used in isolation. Slither may fail to identify vulnerabilities, either due to the lack of specific detectors or whenever certain properties fail to hold after Solidity code is compiled to EVM bytecode. Additionally, due to the random component associated with fuzzing, Echidna may not generate specific edge cases that violate a certain property resulting in false negatives. In order to mitigate these risks, we supplemented our automated testing efforts with a careful manual review of the contracts in scope.

Slither Results

An initial run of Slither on Yeti Finance's smart contracts reported 252 warnings. Some of these warnings correctly reported missing access control checks and arithmetic precision losses. These errors in the codebase were fixed in commit **429557faea0c7ad17cb4a9baea6f9561d8fffe2b,** which also introduced some slight changes to the codebase as well as improved documentation. All remaining warnings were considered false positives.

Echidna Results

Echidna was used to test the expected system properties, simulating interactions between users, liquidators and the core borrowing protocol. In collaboration with the

Yeti team we specified and implemented 50 Echidna properties related to the core borrowing protocol, which are presented in the following table:

Property	Result
After the system is initialized, there is always at least one open trove.	PASSED
The sum of debts from all troves must be the same as the amount of existent YUSD.	PASSED
The sum of total debt cannot be less than the total of SP deposits.	PASSED
YUSD total supply is equal to the sum of the active pool and default pool YUSD debts (total amount of debt in the system).	PASSED
No trove in the sorted list has collateral value less than zero.	PASSED
All troves in the sorted list are active.	PASSED
No trove can have more than *X* different collateral assets.	PASSED
There can never be more than *Y* different active collateral assets.	FAILED (3S-YETI-06)
AP + SP + DP holdings >= (all collateral that has entered into AP - closed trove surplus collateral - withdrawn collateral - liquidated collateral * 0.005 - redeemed collateral)	PASSED
Opening a trove reverts if the trove is already open.	PASSED
Opening a trove reverts if the resulting ICR is lower than 110%.	PASSED
Opening a trove reverts if the resulting net debt is inferior to 2000 YUSD.	PASSED
Opening a trove never reverts if the preconditions are met.	PASSED
Opening a trove adds it to the list of sorted troves.	PASSED
Opening a trove results in a new active trove.	PASSED

Opening a trove mints the correct amount of YUSD to the borrower account.	PASSED
Opening a trove deposits the correct amount of collateral in the trove.	PASSED
Closing a trove reverts if the trove is not open.	PASSED
Closing a trove reverts in recovery mode.	PASSED
Closing a trove reverts if the borrower does not have enough YUSD.	PASSED
Closing a trove never reverts if the preconditions are met.	PASSED
Closing a trove removes it from the list of sorted troves.	PASSED
Closing a trove removes it from the list of liquidatable troves.	FAILED (3S-YETI-02)
Closing a trove results in a new closed trove.	PASSED
Closing a trove burns the correct amount of YUSD from the borrower account.	PASSED
Closing a trove burns the correct amount of YUSD from the gas pool.	PASSED
Closing a trove resets the debt, stakes and rewards.	PASSED
Closing a trove distributes all pending rewards.	PASSED
Adjusting a trove reverts if the adjusted ICR < CCR.	PASSED
Adjusting a trove reverts if the trove is not open.	PASSED
Adjusting a trove by borrowing YUSD increases the borrower's debt.	PASSED
Adjusting a trove by borrowing YUSD increases the borrower YUSD balance.	PASSED
Depositing YUSD into the SP reverts if the deposit amount exceeds the YUSD balance of the depositor.	PASSED
Depositing YUSD into the SP never reverts if the	PASSED

preconditions are met.	
Depositing YUSD into the SP reduces the YUSD balance of the depositor.	PASSED
Depositing YUSD into the SP sends all accrued collateral gains to the depositor.	PASSED
Depositing YUSD into the SP increases the pool deposits of the depositor.	PASSED
Depositing YUSD into the SP increases the total pool deposits.	PASSED
Withdrawing YUSD from the SP reverts if the amount exceeds the deposited balance of the withdrawer.	FAILED (3S-YETI-09)
Withdrawing YUSD from the SP reverts in recovery mode.	PASSED
Withdrawing YUSD from the SP never reverts if the preconditions are met.	PASSED
Withdrawing YUSD from the SP increases the YUSD balance of the withdrawer.	FAILED (3S-YETI-09)
Withdrawing YUSD from the SP sends all accrued collateral gains to the withdrawer.	PASSED
Withdrawing YUSD from the SP decreases the pool deposits of the withdrawer.	FAILED (3S-YETI-09)
Withdrawing YUSD from the SP decreases the total pool deposits.	FAILED (3S-YETI-09)
Liquidating a trove never reverts if the preconditions are met.	PASSED
Liquidating a trove sets its status as closed by liquidation.	PASSED
Liquidating a trove removes it from the list of sorted troves.	PASSED
Liquidating a trove removes it from the list of liquidatable troves.	PASSED
YUSD redemptions never revert if the preconditions are met.	PASSED

Updating liquidatable troves correctly updates internal accounting.	FAILED (3S-YETI-08)

Findings

3S-YETI-01

Open and Adjust Trove operations become impossible for a certain collateral

Id	3S-YETI-01
Severity	Critical
Difficulty	High
Category	Functional Correctness

When there is only one trove holding a specific asset that is about to be liquidated, if the stability pool is also empty, an irrevocable loss of assets happens when the collateral is sent to the DefaultPool contract.

This issue arises because the particular asset cannot be offset against the YUSD in the stability pool, which leads to a redistribution. No one else holds the collateral necessary to receive the redistribution, yet the collateral is still transferred to the DefaultPool contract. As a result, further *openTrove* and *adjustTrove* operations involving the lost collateral are no longer possible, as will be demonstrated in greater detail.

This is only possible in normal mode, as there are no redistributions in recovery mode. When a trove is liquidated in normal mode, the TroveManager *removeStakeTML* method is called first. This function subtracts the *totalStakes* value for each collateral present in the trove.

```
function _removeStake(address _borrower) internal {
    address[] memory borrowerColls = Troves[_borrower].colls.tokens;
    uint256 borrowerCollsLen = borrowerColls.length;
    for (uint256 i; i < borrowerCollsLen; ++i) {
        address coll = borrowerColls[i];
        uint256 stake = Troves[_borrower].stakes[coll];
        totalStakes[coll] = totalStakes[coll].sub(stake);
        Troves[_borrower].stakes[coll] = 0;
    }
}
```

TroveManager.sol#_removeStake

The *totalStakes* value for the collateral in concern is set to zero, as the collateral is not held by any other trove in the system. Following a change to the *totalStakes* variable, redistributions for each collateral occur, as long as staked collateral still exists in the system. Obviously, this will not apply to the mentioned collateral. All collaterals are subsequently transferred to the DefaultPool, from whence they will be dispersed to trove owners upon trove access. Because the collateral was not distributed but was nonetheless sent to the DefaultPool, it is irretrievably lost.

```
function redistributeDebtAndColl(
    IActivePool _activePool,
    IDefaultPool _defaultPool,
    uint256 _debt,
    address[] memory _tokens,
    uint256[] memory _amounts
) external override {
    // (...)
    for (uint256 i; i < tokensLen; ++i) {
        // (...)
        if (totalStakes[token] != 0) {
            // Get the per-unit-staked terms
            uint256 thisTotalStakes = totalStakes[token];
            uint256 CollRewardPerUnitStaked = CollNumerator.div(thisTotalStakes);
            uint256 YUSDDebtRewardPerUnitStaked = YUSDDebtNumerator
                .div(thisTotalStakes.mul(10**(18 - dec)));
            // (...)
            // Add per-unit-staked terms to the running totals
            L_Coll[token] = L_Coll[token].add(CollRewardPerUnitStaked);
            L_YUSDDebt[token] = L_YUSDDebt[token].add(YUSDDebtRewardPerUnitStaked);
            emit LTermsUpdated(token, L_Coll[token], L_YUSDDebt[token]);
        }
    }
    // Transfer coll and debt from ActivePool to DefaultPool
    _activePool.decreaseYUSDDebt(_debt);
    _defaultPool.increaseYUSDDebt(_debt);
    _activePool.sendCollaterals(address(_defaultPool), _tokens, _amounts);
}
```

TroveManager.sol#redistributeDebtAndColl

As previously stated, once this collateral is sent to the DefaultPool contract, *openTrove* and *adjustTrove* operations that utilize the lost collateral are no longer possible.

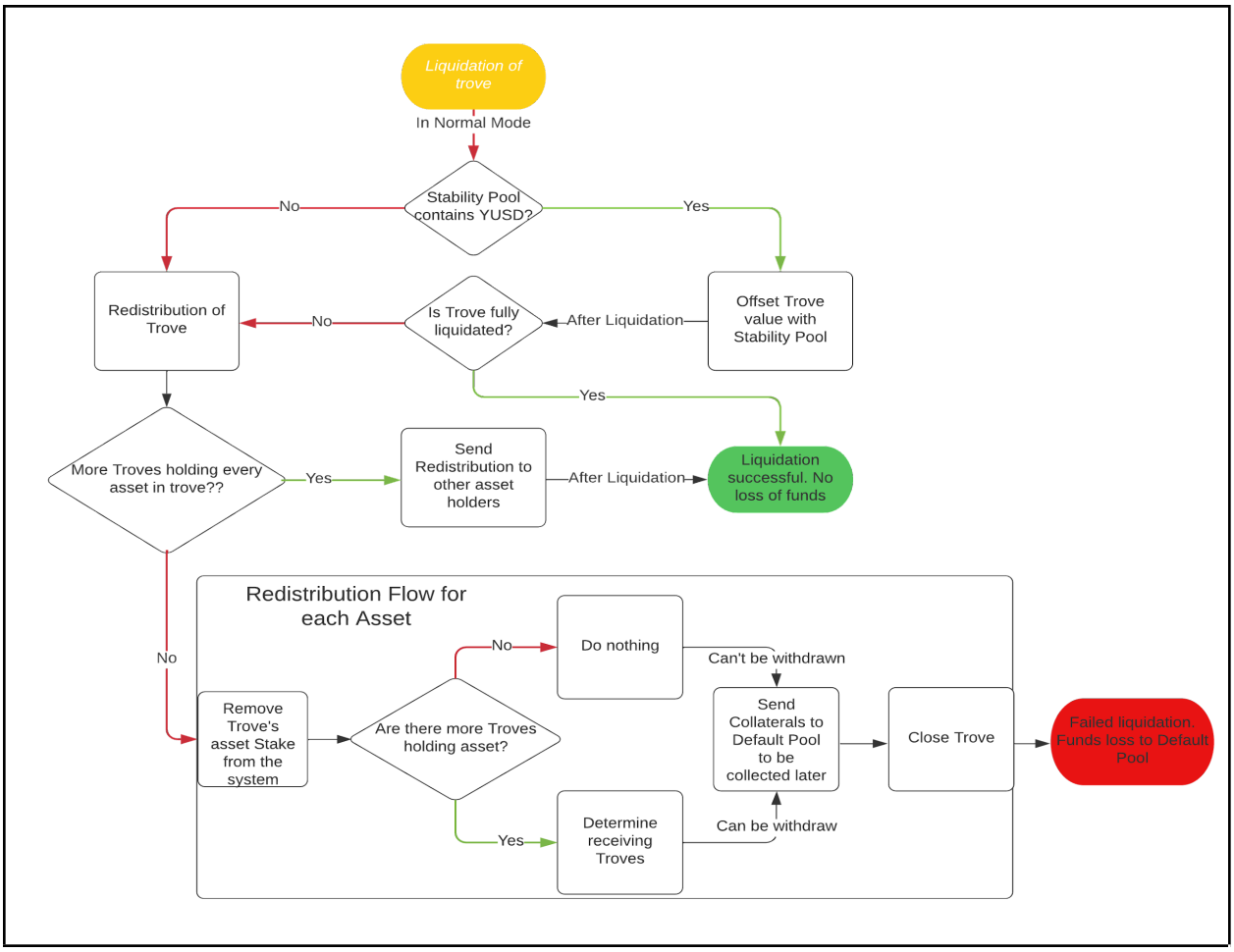
This occurs because both operations call the TroveManager contract function *_computeNewStake*, and the require check within will always fail.

```
function _computeNewStake(address token, uint256 _coll) internal view returns (uint256) {
    uint256 stake;
    if (totalCollateralSnapshot[token] == 0) {
        stake = _coll;
    } else {
        require(totalStakesSnapshot[token] != 0, "TM: stake must be > 0");
        stake = _coll.mul(totalStakesSnapshot[token])
            .div(totalCollateralSnapshot[token]);
    }
    return stake;
}
```

TroveManager.sol#_computeNewStake

TotalCollateralSnapshot[token] will be more than zero, as it reflects the sum of the collateral in the ActivePool and DefaultPool contracts. On the other hand, *totalStakesSnapshot[token]* is equal to 0 since the token is not owned by anybody and so is not staked. This condition will never again be true for this token and all subsequent calls to *openTrove* and *adjustTrove* will revert.

This vulnerability is presented graphically in the computational flow graph.



Redistribution of assets CFG

Recommendation

Several solutions were discussed with the Yeti team in order to fix this edge case. The final solution was implemented in the manner described below.

The function redistributeDebtAndColl checks if totalStakes is 0 before doing a normal redistribution. If it is, an “absorptionColls” list is fetched, which is to be composed of assets deemed safer and highly distributed across all troves, like wavax. Debt is redistributed to those troves as if it was a redistribution for that absorption collateral, according to the weight assigned in YetiController contract. The YetiController contract allows to change the claimAddress and the absorptionColls. If an absorption collateral has a totalStakes of 0, its weight is added to a running total called unAllocatedAbsWeight, and distributed to the next absorption collateral that has a

totalStakes value bigger than 0. If absorptionColls is empty or all of the remaining absorption collaterals also have totalStakes 0, the transaction is reverted.

Status

Fixed in commit **f1cf0d81531b875499a05943ac4db524cd76f941.**

3S-YETI-02

Withdraws from the Stability Pool become impossible

Id	3S-YETI-02
Severity	Critical
Difficulty	Low
Category	Functional Correctness

The TroveManager contract's external function *updateLiquidatableTrove* can add/remove troves that are eligible/not eligible for liquidation from the *liquidatableTroves* mapping and track the amount of liquidatable troves using the *liquidatableTrovesSize* variable.

```
function updateLiquidatableTrove(address _id) external override {
    uint256 ICR = getCurrentICR(_id);
    bool isLiquidatable = ICR < MCR;
    sortedTroves.updateLiquidatableTrove(_id, isLiquidatable);
}
```

TroveManager.sol#updateLiquidatableTroves

The goal of these two variables is to prevent withdrawals from the stability pool when a trove becomes liquidatable. It is simple to detect that withdraws are no longer possible if the *liquidatableTrovesSize* is greater than 1.

```
function _requireNoUnderCollateralizedTroves() internal view {
    ISortedTroves sortedTrovesCached = sortedTroves;
    address lowestTrove = sortedTrovesCached.getLast();
    uint256 ICR = troveManager.getCurrentICR(lowestTrove);
    require(
        ICR >= MCR && sortedTrovesCached.getLiquidatableTrovesSize() == 0,
        "SP:No Withdraw when liquidatable trove"
    );
}
```

StabilityPool.sol#_requireNoUnderCollateralizedTroves

When a trove is liquidated the function *closeTroveLiquidation* from the TroveManager is called. This function also removes the trove from the *liquidatableTroves*, decrementing the *liquidatableTrovesSize* by one as well, by calling the *updateLiquidatableTrove* in the SortedTroves contract.

```
function closeTroveLiquidation(address _borrower) external override {
    _requireCallerIsTML();
    // Remove from liquidatable trove if it was there.
    sortedTroves.updateLiquidatableTrove(_borrower, false);
    return _closeTrove(_borrower, Status.closedByLiquidation);
}
```

TroveManager.sol#closeTroveLiquidation

However, it is possible that a trove is added to the *liquidatableTroves* mapping and before there is a chance to liquidate it, the borrower closes the trove by calling the function *closeTrove* from the BorrowerOperations, which in turn calls the TroveManager contract's *removeStakeAndCloseTrove* function.

```
function removeStakeAndCloseTrove(address _borrower) external override {
    _requireCallerIsBorrowerOperations();
    _removeStake(_borrower);
    _closeTrove(_borrower, Status.closedByOwner);
}
```

TroveManager.sol#removeStakeAndCloseTrove

The trove was closed but it was not removed from the *liquidatableTroves* variable, while still effectively removing it from the sortedTroves list.

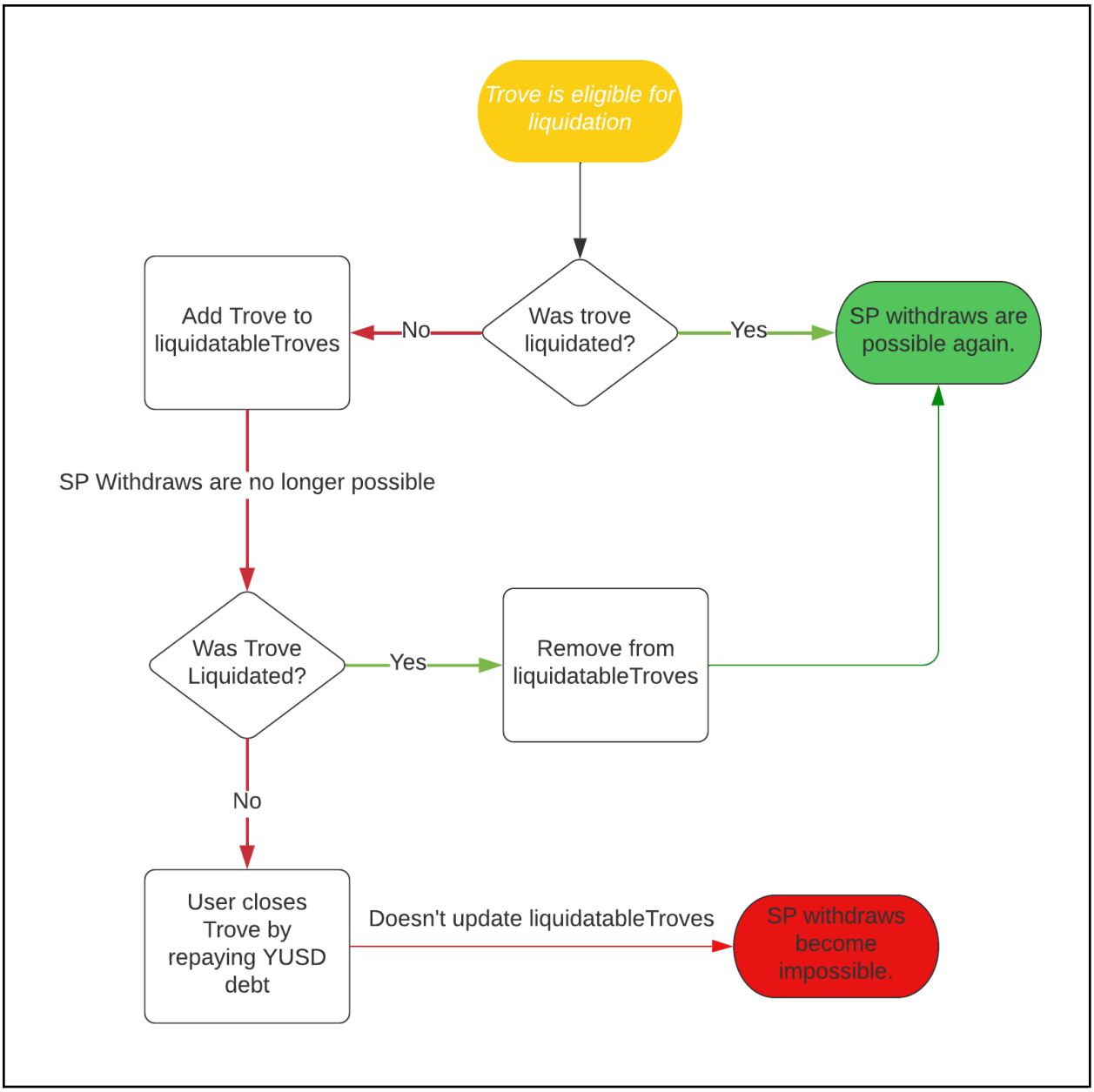
Withdrawals from the SP are never possible from this point forward. This occurs because removing the closed trove from the *liquidatableTroves* variable is impossible since the trove in question is no longer in the sortedTroves list, which is a require in the function *updateLiquidatableTrove* in the SortedTroves contract.

```
function updateLiquidatableTrove(address _id, bool _isLiquidatable) external override {
    _requireCallerIsTroveManager();
    require(contains(_id), "SortedTroves: Id not found");
    if (_isLiquidatable) {
        // If liquidatable and marked not liquidatable, add to list
        if (!liquidatableTroves[_id]) {
            _insertLiquidatableTrove(_id);
        }
    } else {
        // If not liquidatable and marked liquidatable, remove from list
        if (liquidatableTroves[_id]) {
            _removeLiquidatableTrove(_id);
        }
    }
}
```

SortedTroves.sol#updateLiquidatableTrove

This implies that it is not possible to bring the *liquidatableTrovesSize* value back to zero, effectively reverting every SP withdraw call. Because a significant portion of the system's debt is placed in the SP, which is now locked, there will be insufficient YUSD for people to close their troves, therefore the collateral in the troves is also indirectly locked. Note that all of the steps mentioned in this vulnerability which lockup up the funds can be done atomically by any user.

This vulnerability is presented graphically in the computational flow graph.



SP Withdrawals CFG

Recommendation

Our recommendation is to change the location of the *updateLiquidatableTrove* call from the *closeTroveLiquidation* to the *_closeTrove* call, effectively checking every time a trove is closed if it is in the *liquidatableTrove* variable.

```
function _closeTrove(address _borrower, Status closedStatus) internal {
    require(
        closedStatus != Status.nonExistent && closedStatus != Status.active,
        "Status must be active and exists"
    );
     // 3Sigma recommendation
    sortedTroves.updateLiquidatableTrove(_borrower, false);
    // (...)
    sortedTroves.remove(_borrower);
}
```

TroveManager.sol#_closeTrove

Status

Fixed in commit **de024893076ea1cf7590693fee54e0c983f5f5f5**.

3S-YETI-03

Incorrect YetiController modifier

Id	3S-YETI-03
Severity	Low
Difficulty	N/A
Category	Access Control

YetiController.updateMaxCollsinTrove had the incorrect modifier *onlyOwner*.

Recommendation

Changing the modifier to *oneWeekTimelock*.

Status

Fixed in commit **22770adc002c8dd741c7f9254233de8d8b8b366e.**

3S-YETI-04

ThreePieceWiseLinearFeeCurve doesn't check if slopes are strictly increasing

Id	3S-YETI-04
Severity	Low
Difficulty	N/A
Category	Functional Correctness

The contract ThreePieceWiseLinearFeeCurve computes the backing fee depending on the overall backing percentage of that asset in the system. The more the backing fee, the greater the fee rise should be. This indicates that the slope of the first line should be less than the slope of the second, and the slope of the second should be less than the slope of the third. The contract contains no explicit require clauses that ensure that human mistakes are not possible when setting these values.

```
function adjustParams(
    string memory _name,
    uint256 _m1,
    uint256 _b1,
    uint256 _m2,
    uint256 _cutoff1,
    uint256 _m3
    uint256 _cutoff2,
    uint _dollarCap,
    uint _decayTime
) external onlyOwner {
    require(_cutoff1 <= _cutoff2, "Cutoffs must be increasing");
    // (...)
}
```

ThreePieceWiseLinearFeeCurve.sol#adjustParams

Recommendation

Add require clauses that disallow the described behavior.

Status

Fixed in commit **22770adc002c8dd741c7f9254233de8d8b8b366e.**

3S-YETI-05

Redemptions of undercollateralized troves is possible

Id	3S-YETI-05
Severity	Low
Difficulty	Low
Category	Functional Correctness

The function *reedemCollateral* iterates across the sorted troves to identify the trove with the minimum RICR eligible for redemption. When it discovers the trove, it redeems against that trove. If there is surplus YUSD after the initial redemption, it iterates from there redeeming to the succeeding troves until all value has been redeemed.

Redemptions from Troves are permitted only if the trove is not under collateralized. To verify this, the ICR of the trove must be greater than the MCR. However, when the function is attempting to determine the lowest RICR, it compares the RICR to the MCR rather than the ICR. It is possible that the chosen trove has an ICR smaller than the MCR and so it should not be redeemable.

```
function _isValidFirstRedemptionHint(
    ISortedTroves _sortedTroves,
    address_firstRedemptionHint
) internal view returns (bool) {
    if (
        _firstRedemptionHint == address(0) ||
        !_sortedTroves.contains(_firstRedemptionHint) ||
        troveManager.getCurrentRICR(_firstRedemptionHint) < MCR
    ) { return false; }

    address nextTrove = _sortedTroves.getNext(_firstRedemptionHint);
    return nextTrove == address(0) || troveManager.getCurrentICR(nextTrove) < MCR;
}
```

TroveManagerRedemptions.sol#_isValidFirstRedemptionHint

By providing the correct address of the trove with the lowest RICR, the function *_isValidFirstRedemptionHint* is utilized in place of searching for the trove.
This function's goal is to save gas for the function's caller.

The comparison done here suffers from the same issue, since it compares the RICR to the MCR rather than the ICR.

Recommendation

Compare the ICR with the MCR instead of the RICR.

Status

Fixed in commit **22770adc002c8dd741c7f9254233de8d8b8b366e.**

3S-YETI-06

No explicit limit on number of system collaterals

Id	3S-YETI-06
Severity	Low
Difficulty	N/A
Category	Functional Correctness

The quantity of gas required to liquidate a trove is determined primarily by two factors: the number of distinct collaterals in the system and the number of distinct collaterals in the trove. To ensure that no liquidation exceeds the block limit of 8M gas, these two variables must always be smaller than a certain value.

```
function addCollateral(
    address _collateral,
    uint256 _safetyRatio,
    uint256 _recoveryRatio,
    address _oracle,
    uint256 _decimals,
    address _feeCurve,
    bool _isWrapped,
    address _routerAddress
) external override onlyOneWeekTimelock {
    checkContract(_collateral);
    checkContract(_oracle);
    checkContract(_feeCurve);
    checkContract(_routerAddress);
    // If collateral list is not 0, and if the 0th index is not equal to this collateral,
    // then if index is 0 that means it is not set yet.
    require(
      _safetyRatio < 11e17,
      "Safety Ratio must be less than 1.10"
    ); //=> greater than 1.1 would mean taking out more YUSD than collateral VC
    require(_recoveryRatio >= _safetyRatio, "Recovery ratio must be >= safety ratio");
    // (...)
}
```

YetiController.sol#addCollateral

The maximum number of distinct assets in the trove is implemented, but not the maximum number of distinct assets in the system. Rather than that, the Yeti team controls this second variable, as it is the team that whitelists collaterals for inclusion in the system. It is possible for the Yeti team to whitelist more collaterals than it should, resulting in a total gas cost for liquidation exceeding 8M.

Recommendation

To implement an explicit requires clause that disallows whitelisting collaterals if the total number of collaterals in the system is bigger than a certain value.

Status

Fixed in commit **22770adc002c8dd741c7f9254233de8d8b8b366e.**

3S-YETI-07

Improper naming of require function

Id	3S-YETI-07
Severity	None
Difficulty	N/A
Category	Best Practice

The *requireNoUnderCollateralizedTroves* in the StabilityPool *withdrawFromSP* function is called to ensure that no liquidatable troves are present in the system at the time of withdrawal.

```
function _requireNoUnderCollateralizedTroves() internal view {
    ISortedTroves sortedTrovesCached = sortedTroves;
    address lowestTrove = sortedTrovesCached.getLast();
    uint256 ICR = troveManager.getCurrentICR(lowestTrove);
    require(
        ICR >= MCR && sortedTrovesCached.getLiquidatableTrovesSize() == 0,
        "SP:No Withdraw when liquidatable trove"
    );
}
```

StabilityPool.sol#_requireNoUnderCollateralizedTroves

It examines the lowest RICR trove of the *sortedTroves* and compares its ICR to the MCR to determine whether or not this trove is liquidatable.
It should be noted that the requirement does not take into account the system's recovery mode liquidation conditions. This is a protocol design decision, so it is still possible to withdraw YUSD in recovery mode to repay the debt of troves. It is not clear by the name of the function *_requireNoUnderCollateralizedTroves* that it should only check if there are liquidatable troves in normal mode.

Recommendation

Change to a name that more accurately describes the function behavior.

Status

Fixed in commit **22770adc002c8dd741c7f9254233de8d8b8b366e.**

3S-YETI-08

Improper naming of updateLiquidatableTrove function

Id	3S-YETI-08
Severity	None
Difficulty	N/A
Category	Bad Practice

The *updateLiquidatableTrove* function in the TroveManager contract checks if a trove is liquidatable or not and adds/removes the trove from the liquidatableTroves by calling the *updateLiquidatableTrove* function in the SortedTroves contract. This is a protocol design decision to allow for the withdrawal of YUSD in recovery mode to repay troves' debt. The name of the function _requireNoUnderCollateralizedTroves does not indicate that it should only check for liquidatable troves in normal mode.

```
function updateLiquidatableTrove(address _id) external override {
    uint256 ICR = getCurrentICR(_id);
    bool isLiquidatable = ICR < MCR;
    sortedTroves.updateLiquidatableTrove(_id, isLiquidatable);
}
```

TroveManager.sol#updateLiquidatableTrove

The variable *isLiquidatable* does not take into account the system's recovery mode liquidation conditions. This is a protocol design decision to allow for the withdrawal of YUSD in recovery mode to repay troves' debt. The name of the variable *liquidatableTroves* and function *updateLiquidatableTroves* don't indicate that it should only check for liquidatable troves in normal mode.

Recommendation

Change to a name that more accurately describes the function behavior.

Status

Fixed in commit **22770adc002c8dd741c7f9254233de8d8b8b366e.**

3S-YETI-09

Unexpected StabilityPool YUSD withdrawal amounts

Id	3S-YETI-09
Severity	None
Difficulty	N/A
Category	Best Practice

The *_withdrawFromSP* function in the StabilityPool was found to violate multiple of our fuzzing properties, namely as it doesn't revert whenever it's called with an *_amount* parameter that exceeds the caller's deposit or equal to zero. Manual code review revealed that this behavior was intentional, and even documented in the code, as any *_amount* surplus is internally capped to the total compounded deposits of the caller, and a zero _amount withdrawal simply collects any pending collateral gains without actually withdrawing YUSD. Nonetheless, we find the decision of internally capping the withdrawal amount inconsistent with the rest of the codebase.

```
function _withdrawFromSP(uint256 _amount)
    internal
    returns (address[] memory assets, uint256[] memory amounts)
{
    if (_amount != 0) {
        _requireNoUnderCollateralizedTroves();
    }
    // (...)
    uint256 compoundedYUSDDeposit = getCompoundedYUSDDeposit(msg.sender);
    uint256 YUSDtoWithdraw = YetiMath._min(_amount, compoundedYUSDDeposit);
    // (...)
    _sendYUSDToDepositor(msg.sender, YUSDtoWithdraw);
    // (...)
}
```

StabilityPool.sol#_withdrawFromSP

Recommendation

Replace the internal capping of the withdrawal amount with a requirement which reverts whenever the amount exceeds the total compounded deposits of the caller.

Status

Acknowledged by the Yeti team.