PUBLIC

Code Assessment

of the Uncap Finance Smart Contracts

September 26, 2025

Produced for

by

Contents

1 Executive Summary

Dear Uncap Finance team,

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

Uncap Finance implements a BTC backed stablecoin on Starknet. It forks the Liquity v2 stablecoin mechanism, re-implementing it in the Cairo language.

The most critical subjects covered in our audit are adherence to the original codebase, access control, and correctness of the new features. In the latest reviewed version, all issues have been resolved and security regarding all the aforementioned subjects is high.

The general subjects covered are upgradeability and secure integration with oracles. After the redesign of WBTCPriceFeed, security regarding all the aforementioned subjects is high.

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

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

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

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

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

Critical-Severity Findings	0
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	5
• Code Corrected	5
Low-Severity Findings	5
• Code Corrected	3
• Risk Accepted	2

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Uncap Finance repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	25 Aug 2025	edb1ccb7a4bb8b09b4edfe7e704017a3df138f32	Initial Version
2	07 Sep 2025	caefb99a727561ee15596c794d00edc12ade53af	WBTC Price Feed
3	16 Sep 2025	1710892bf98ec4f71f2c17669fa21e94a3506886	First Round of Fixes
4	18 Sep 2025	71afb890dde8dfee9713bbaa40989c49e3b696b9	Oracle & Cap Fixes
5	25 Sep 2025	3d66745fd58dd1f43218a62ce59393c85d98512a	Parameter Updates

For the Cairo contracts, the compiler version 2.12.0 was chosen. At the time of this review (September 2025), Starknet v0.14.0 was live on mainnet. This review cannot account for future changes and possible bugs in Starknet and its libraries.

The review assumes Liquity V2, considered at commit `2bdffa63097cbd7c8cc5457db3021e3260a1e59c`, is secure and focus on the implementation difference of the following files:

```
src/
    lib.cairo
    USDU.cairo
    CollateralRegistry.cairo
    ActivePool.cairo
    CollSurplusPool.cairo
    AddressesRegistry.cairo
    DefaultPool.cairo
    SortedTroves.cairo
    BorrowerOperations.cairo
    GasPool.cairo
    StabilityPool.cairo
    TroveNFT.cairo
    trove/
        BatchManager.cairo
        LiquidationManager.cairo
        TroveManager.cairo
        RedemptionManager.cairo
        TroveManagerEventsEmitter.cairo
    utils/
        constants.cairo
        default_address.cairo
        fast_power.cairo
        i257.cairo
        math.cairo
```

```
price_feeds/
    WBTCPriceFeed.cairo
wrappers/
    CollateralWrapper.cairo
components/
    AddRemoveManagers.cairo
    UncapBase.cairo
    UpgradeableBase.cairo
```

2.1.1 Excluded from scope

Any file not listed above is excluded from the scope. In particular, libraries, dependencies are out of scope and assumed to work as documented.

Uncap Finance implements equivalent accountings as Liquity V2 with added features. The review assumes Liquity V2 is secure and is focused on the semantic and functional equivalence, and verifying the new features. For more information of Liquity V2, please consult Liquity V2 Review.

The configuration and parameterization of the system is out of scope.

2.2 System Overview

Uncap Finance is a fork of Liquity V2 on Starknet. Since it runs on Starknet, the Liquity V2 contracts have been rewritten in Cairo, the Rust based language supported by Starknet.

New features have been implemented such as a per-branch debt cap, to limit the risk exposure to certain collaterals, addition of new branches, upgradeability of all contracts, modifiable MCR, sponsoring of the liquidation refunds, collateral wrapper. The following global parameters have been modified: `MIN_DEBT`, `STRK_GAS_COMPENSATION`, `COLL_GAS_COMPENSATION_CAP`.

Uncap Finance offers USDU, a stablecoin pegged to the dollar backed by BTC. The stablecoin logic follows that of Liquity V2, closely reimplemented in Cairo.

Uncap Finance implements a few novel features:

- New collaterals can be onboarded by a trusted admin.
- Per-collateral debt caps can be configured to limit the risk exposure.
- Contracts have been made upgradeable.
- The Minimum Collateralization Ratio can be lowered by the admin once it has been set.
- Gas Compensations are paid in STRK, the native token on Starknet, and they can be sponsored by Uncap Finance when opening a trove.
- A token wrapper has been implemented to extend the number of decimals of collaterals to 18 in case they have fewer.
- The `MIN_DEBT`, `STRK_GAS_COMPENSATION`, and `COLL_GAS_COMPENSATION_CAP` constants have been modified.
- The TroveManager contract has been refactored into 5 contracts: TroveManager, BatchManager, LiquidationManager, RedemptionManager, TroveManagerEventEmitter.
- Price feed for WBTC.

2.2.1 Refactoring

To overcome the contract size limit on Starknet, the TroveManager has been split into several contracts:

- TroveManager contains most of the state and functions to open, modify, and close troves, to join and leave batches, and to liquidate troves.
- However the internal accounting logic related to batches, and the state of batches has been moved into the BatchManager contract, and the logic for liquidations has been moved into the LiquidationManager.
- The branch entry point for redemption has been moved from the TroveManager to the RedemptionManager.
- RedemptionManager, BatchManager, and LiquidationManager all need to access and modify the state held in the TroveManager and BatchManager. For this purpose, permissioned external functions to modify the state of the TroveManager and the BatchManager have been created. These functions can only be called by the contracts that have been factored out of the original TroveManager to maintain the original logic.
- Events from these 4 contracts are emitted by the TroveManagerEventEmitter contract, so that they can be indexed at a single address.

2.2.2 Parameter Changes

Due to the lower gas costs on Starknet, `MIN_DEBT` has been lowered to 200. This allows opening smaller troves. The `MIN_DEBT` parameter is necessary to prevent adversarial actors from griefing the system by opening small positions. Redemptions operate on each position individually in a loop, and small positions require the same amount of gas to redeem as big positions. This may make redemptions unprofitable if `MIN_DEBT` is too low. The new `MIN_DEBT` is lower than on Liquity because lower gas costs mean smaller positions can be redeemed profitably.

Gas compensation is not in ETH but in STRK, and the `STRK_GAS_COMPENSATION` amount has been defined as 10 STRK. `COLL_GAS_COMPENSATION_CAP` has been set accordingly to the collateral type, which is bitcoin, to 0.1 BTC.

The redemption fee floor has been set to `0.4%` given that the price feeds' deviation from the market price is lower than on Ethereum mainnet.

2.2.3 Addition of New Branches

Unlike Liquity, where the collaterals are immutably set at system deployment time in the CollateralRegistry contract, in Uncap Finance the CollateralRegistry holds a dynamic list of collaterals, which can be extended by the owner of the contract.

The CollateralRegistry owner can add a new collateral by independently deploying the contracts for the new branch (a branch is the set of contracts that handle USDU issuance for a given collateral), and the new branch is added to the system by registering the AddressesRegistry for the new branch in the CollateralRegistry through the `add_collateral` function. The USDU token must also be configured to allow burning and minting of tokens from the addresses of the new branch: It exposes the `add_branches_addresses` function that allows the CollateralRegistry owner to register these addresses as privileged in the token contract.

Collaterals with fewer than 18 decimals must be wrapped in the CollateralWrapper contract, which issues equivalent tokens but with precision extended to 18 decimals.

2.2.4 Upgradeability

All contracts have been made upgradeable through UpgradeableBaseComponent, a component which exposes the public function `upgrade()` (except for the CollateralRegistry contract itself and the CollateralWrapper contract which implements upgradeability differently). The owner of the CollateralRegistry can call the `upgrade()` function of each contract. The upgrade function uses the native upgradeability mechanism of Starknet (system call `replace_class_syscall`) to replace the code the contract is using.

The CollateralWrapper contract implements upgradeability differently, by using UpgradeableComponent of OpenZeppelin. The mechanism implemented is similar, however the owner is defined per contract, and is not just the owner of the CollateralRegistry contract. As such, the CollateralWrapper contracts could have different owners than the system wide admin.

2.2.5 Debt Cap

Branch specific Debt Caps can be configured by the admin, to reduce the system's exposure to single collaterals. Reaching the Cap prevents opening new troves (positions), or withdrawing more USDU from existing troves. However, the cap be exceeded by interest accrual on existing debt, and by accrual of upfront fees generated when adjusting the interest rate of an existing trove.

2.2.6 Configurable Collateralization Ratio

The Minimum Collateralization Ratio of each branch can be modified by the admin, with the `decrease_mcr` function defined in the AddressesRegistry. The new value needs to be smaller than the current one, such that the health of existing troves can only be improved through this function, preventing the admin from triggering liquidations.

2.2.7 Price Feeds

Uncap Finance implements a price feed for WBTC using the Pragma oracle as a primary source and Chainlink as a backup source.

When fetching prices for redemptions, the maximum of the WBTC and BTC prices is used, up to a deviation of 1% after which the WBTC price is used.

For each price, WBTC or BTC, Pragma is first queried. If it fails, because not enough sources are aggregated (MIN_NUM_SOURCES is 5), the price is zero, or the price is older than STALENESS_THRESHOLD (set to 24 hours), the Chainlink backup is used. If Chainlink fails because the price is zero or the price is stale (same STALENESS_THRESHOLD), then Pragma is used anyways if its response was valid but had fewer sources than MIN_NUM_SOURCES. If Pragma had failed for another reason, then an oracle failure is reported, the oracle is shut down and the shutdown is propagated to the system.

2.2.7.1 Pragma

Pragma is a Starknet specific oracle provider. It provides prices for multiple pairs, including WBTC/USD and BTC/USD. Each price pair has multiple sources, where the prices are queried (e.g. Defillama, Binance, Coinbase). Each source has one or more publishers, which are Starknet addresses allowed to push price updates on chain.

When Uncap Finance queries the price with median as the aggregation mode, the following computation will be performed:

- Entries reported by publishers are filtered by a freshness check. Namely, entries over 1 hour older than the most recent entry are discarded and not used in the following aggregation. Notably, sources without any fresh entry will be excluded from the aggregation.
- Within each source, the prices provided by different publishers are aggregated and the source price is computed as their median.

- The final price is computed as the median over all sources.
- The price resulting from the median aggregation has a timestamp associated with it, which is the most recent value that took part in the median. The number of sources aggregated is also one of the returned values.

Sources could each have different publishers, however in practice Pragma as an organization is also the publisher of most sources. Since they publish more than half of the sources, they are therefore able to arbitrarily move the median. Using the Pragma oracles therefore implies trusting Pragma to publish correct values.

The Pragma contract is upgradable and found at address:

- 0x02a85bd616f912537c50a49a4076db02c00b29b2cdc8a197ce92ed1837fa875b .

2.2.7.2 Chainlink

Chainlink feeds on Starknet operates similarly to how it operates on EVM chains. A publisher transmits new price reports on chain, which consist in a sorted list of values. The list has been validated and signed by a quorum of chainlink nodes. The signatures are verified on-chain, and the median value is computed by taking the middle element of the list of prices, which is sorted. WBTC and BTC price feeds are configured to have a deviation threshold of 0.1%, which is the price change after which Chainlink posts an update.

Chainlink contracts are upgradable. The owner of the Chainlink contract can also make the data feeds permissioned, such that only white-listed addresses can query `latest_round_data()` and similar functions.

2.3 Trust Model

The following roles exist in the system, with the relative trust model:

- The Owner of the CollateralRegistry contract: **fully trusted**. It can upgrade other contracts (except for the CollateralWrappers), and add new branches. In the worst case, it can upgrade contracts to malicious implementations to seize the collateral or mint USDU for free.
- The Owners of the CollateralWrapper contracts: **fully trusted**. In the worst case, they can upgrade the CollateralWrappers to seize the underlying collateral.
- Deployer of the contracts: **fully trusted**. Added in Version 3, it is a one-time role set in the constructor upon deployment, who is expected to initialize the contracts with correct parameters.

Further, the following assumptions are made on external entities:

- The Pragma Publisher Registry admin: **fully trusted**. It can configure the publishers that are allowed for sources hence in the worst case manipulate the price.
- The Pragma Oracle admin: **fully trusted**. In the worst case, it can configure Publisher Registry hence perform a similar price manipulation. Further, it can upgrade the contract to directly manipulate the price.
- The Pragma Publishers: **semi-trusted**. In the worst case, they can manipulate the prices that they are allowed to publish, hence tweak the median price computation in some extent. **Note** regarding the specific oracles Uncap Finance uses, Pragma is the publisher of over 50% of the sources: 6/10 for WBTC/USD and 10/12 for BTC/USD, hence Pragma is **fully trusted** to be honest.
- The Starknet sequencer is trusted to not censor transactions, as there is no force inclusion mechanism for transactions on Starknet.
- The Chainlink AggregatorProxy owner: **fully trusted**. In the worst case, it can switch the aggregator contract or upgrade the Chainlink oracle to manipulate the price. It can also enable the access control and block Uncap Finance from reading the price.

- The Chainlink price reporters are **partially trusted**, as long as fewer than one third are malicious or faulty, the reported prices cannot be manipulated.

Collateral tokens used in the system are expected to have 18 decimals or being wrapped with the CollateralWrapper. They should not implement fee on transfer, transfer hooks, or rebasing mechanism.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	2

5.1 Compromised Collateral Allows Unlimited Value Extraction Despite Debt Cap

Design Low Version 1 Risk Accepted

CS-UNCAP-001

Since the upfront fee is excluded from the debt cap check, it is possible to elude the debt cap by repeatedly paying upfront fees, for example by adjusting the interest rate. If the collateral can be freely minted by an attacker, the attacker can open a trove with a very high amount of collateral, and trigger the upfront fee in a loop hence blowing up the total debt on this branch. Profit can be extracted by being a receiver of the upfront fees in the Stability Pool, which are payed in USDU. The attacker can therefore extract an unbounded amount of USDU and this can eventually lead to USDU depegging for all branches.

Risk accepted:

Uncap Finance has accepted the risk and stated:

```
This will be mitigated by cautiously selecting our collaterals.
```

5.2 Sponsor Can Be Drained

Design Low Version 1 Risk Accepted

CS-UNCAP-002

The sponsor provides `STRK_GAS_COMPENSATION` to every new trove as a Gas Compensation for a potential future liquidator. Depending on the cost of self-liquidations, it might be profitable for an attacker to open many troves, and liquidate them to earn the `STRK_GAS_COMPENSATION`. Assuming `MIN_DEBT` is 200, `STRK_GAS_COMPENSATION` is 10, then opening a trove needs at least 200 USDU of debt, of which 5% is paid, during a liquidation, in collateral as a liquidation discount to the stability pool.

The profitability conditions for an attacker are, disregarding gas costs:

```
10 STRK + 0.75 * attacker_share_of_stability_pool * MIN_DEBT * liquidation_discount
    >= MIN_DEBT * upfront_fee_rate + MIN_DEBT * liquidation_discount
```

Because part of the liquidation is recovered by the attacker if they have share in the stability pool. For the aforementioned parameter values, disregarding the upfront fee (small), we have:

```
10 * STRK price + 0.75 * attacker_share_of_stability_pool * 200 * 0.05 >= 200 * 0.05
```

So the attacker can extract value if they own more than `(10 - STRK_PRICE * 10)/(0.75 * 10)` as a ratio of the stability pool. Currently STRK_PRICE is 0.13, so the profitability cannot be reached, since it would require owning a ratio of the stability pool greater than 1. However the strategy might already become profitable if STRK is priced at $0.25, in case the attacker controls 100% of the stability pool. In case STRK price reaches $1, the strategy becomes profitable without any stake in the stability pool.

Risk accepted:

Uncap Finance has accepted the risk and stated:

```
We will monitor sponsor's balance as well STRK price.
```

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	1

- Unprotected Initializer Code Corrected

Medium-Severity Findings	5

- Accidental Shutdown May Be Triggered After Long Sequencer Outage Code Corrected
- Oracle Failure Cannot Trigger Shutdown Code Corrected
- Cap Check May Block Legitimate Operations Code Corrected
- Incorrect Position Lookup in Sorted Trove Code Corrected
- Missing kick_from_batch Functionality Code Corrected

Low-Severity Findings	3

- Shutdown Price May Be Over-Estimated Code Corrected
- Dust May Be Locked in Collateral Wrapper Code Corrected
- Inconsistent Cap Check Code Corrected

Informational Findings	5

- Gas Optimization Code Corrected
- Missing Getters for deposit_snapshots Code Corrected
- Redundant Code Code Corrected
- Stale Cairo Compiler Configuration Code Corrected
- Typos and Misnamed Functions & Variables Code Corrected

6.1 Unprotected Initializer

Security High Version 1 Code Corrected

CS-UNCAP-017

Most contracts of Uncap Finance are upgradeable and an initializer pattern is used to setup the critical configurations such as contract wiring and parameters after deployment. However, the initializers are not protected. Hence, a malicious user can backrun the contract deployment and initialize the contracts with wrong or malicious configurations. Consequently:

- One can DoS the initialization of the contracts.
- One can become the owner of the deployed contracts by passing a manipulated `addresses_registry` address.
- One can upgrade the contracts with a backdoor, which can be leveraged to exploit the protocol if it is not later recognized by the legitimate deployer.

Code corrected:

In Version 3, a trusted `deployer` address is set at construction time in the contracts. Only this `deployer` address is allowed to initialize the contracts, therefore preventing backrunning of deployments.

6.2 Accidental Shutdown May Be Triggered After Long Sequencer Outage

Security Medium Version 2 Code Corrected

CS-UNCAP-018

WBTCPriceFeed uses Pragma oracle to get the median price. In pragma, multiple sources can be configured for one asset, each source may have multiple publishers to publish multiple entries. These publishers will send individual transactions to publish the price entries asynchronously. Upon a read request (`get_data_median()`), onchain aggregation will be performed over all available entries:

1. Fresh entries will be filtered out based on the latest entry and the `BACKWARD_TIMESTAMP_BUFFER` (currently set to 1 hour). Namely, any price that is 1 hour older than the most recent one (or now) will be excluded from the aggregation.
2. The price of each source is computed as the median of all its fresh entries.
3. The price of this asset is computed as the median of all sources, and the `num_sources_aggregated` is the number of sources that contain fresh entries.

In Uncap Finance, a shutdown will be triggered in case an oracle failure is detected. In WBTCPriceFeed, the oracle failure may occur in case there is insufficient sources aggregated in the Pragma price response:

```
let not_enough_sources = response.num_sources_aggregated < MIN_NUM_SOURCES;

let oracle_failure_detected = price_is_zero
    || incorrect_decimals
    || is_stale_price
    || not_enough_sources;
```

In case of a sequencer outage longer than 1 hour, when the sequencer comes back and the first price entry is updated, all the other entries, which are now more than 1 hour older than the most recent one, will be excluded from the price aggregation, hence `response.num_sources_aggregated` will be 1. As a result, given a `MIN_NUM_SOURCES` greater than 1, an oracle failure could be triggered to accidentally shutdown Uncap Finance after the sequencer outage.

Code corrected:

In Version 4 the WBTCPriceFeed is redesigned and a Chainlink oracle is added as a fallback in case of Pragma oracle failure. In case an insufficient number of sources were aggregated in the Pragma oracle after a network outage, the Chainlink price will be used with a 24 hour staleness threshold. Furthermore, even if a Chainlink failure is detected, the Pragma oracle price will still be regarded valid if Pragma failure is only due to insufficient sources.

6.3 Oracle Failure Cannot Trigger Shutdown

Design | Medium | Version 2 | Code Corrected

CS-UNCAP-012

Protocol shutdown can be triggered by calling `shutdown()` or `shutdown_from_oracle_failure()`. The latter is restricted to be called by the `price_feed`. Further, as the specs suggested, `shutdown()` expects the `price_feed` to call `shutdown_from_oracle_failure()` during `fetch_price()`. However, the `price_feed` never calls `shutdown_from_oracle_failure()`. As a result, shutdown can never be triggered from oracle failure. In addition, this means the `price_feed` itself will shutdown and serve the last valid price, but the branch will not shutdown and will keep operating with a stuck price.

Code corrected:

In Version 4 WBTCPriceFeed has been corrected to trigger a shutdown on BorrowerOperations during the call of `fetch_price()` or `fetch_redemption_price()` if failure is detected.

6.4 Cap Check May Block Legitimate Operations

Design | Medium | Version 1 | Code Corrected

CS-UNCAP-010

The Debt Cap is designed to limit the maximum debt that can be generated on individual branches to restrict the risk that a single asset poses on USDU peg. It is always enforced in `_adjust_trove()` with the following check.

```
fn require_cap_not_exceeded(self: @ContractState, usdu_amount: u256) {
    if let Some(cap) = self.uncap_base.get_addresses_registry().get_cap() {
        let total_debt = self.uncap_base.get_entire_branch_debt();
        assert(cap >= total_debt + usdu_amount, 'BO: Cap exceeded');
    }
}
```

Consequently, if the total branch debt has reached the cap, the following operations are blocked, even if they do not increase the debt: `add_coll()`, `adjust_trove()`, `repay_usdu()`, `adjust_zombie_trove()`, `withdraw_coll()`, `withdraw_usdu()`.

Code corrected:

In Version 4 the cap check in `_adjust_trove()` has been corrected, and is only enforced if there is a debt increase regardless of the fees and interest.

6.5 Incorrect Position Lookup in Sorted Trove

Correctness | Medium | Version 1 | Code Corrected

CS-UNCAP-006

Function `_find_insert_position()` is expected to find the correct position to insert the trove with the input annual interest rate. In case the hints were outdated but the correct position is still between two hints, it will call `descend_and_ascend_list()` to find the correct location. However, function `descend_and_ascend_list()` returns a mix of `descent_pos` and `ascent_pos`, hence may insert

the trove to a wrong position. Consequently, the troves may not be correctly sorted according to their annual interest rate, and the redemption order may be manipulated.

```
while (true) {
    if (self
        .descend_one(trove_manager_address, annual_interest_rate, ref descent_pos)) {
        return (descent_pos.prev_id, ascent_pos.next_id);
    }
    if (self.ascend_one(trove_manager_address, annual_interest_rate, ref ascent_pos)) {
        return (descent_pos.prev_id, ascent_pos.next_id);
    }
}
```

Code corrected:

The issue is corrected in Version 3 of the code by returning the correct variables in the right order.

6.6 Missing `kick_from_batch` Functionality

Design Medium Version 1 Code Corrected

CS-UNCAP-011

External function `kick_from_batch()` is missing in the Uncap implementation of BorrowerOperations. `kick_from_batch()` is required to handle a couple of edge-cases involving inflated debt share values in batches:

- To avoid rounding attacks against batch shares, new debt shares are not minted in batches when the share value becomes more than 1e9 debt. This prevent attackers from minting bold for free by exploiting the shares rounding (see issue CS-BOLD-001 of ChainSecurity Bold Review).
- But this also introduces a potential pitfall, where an unredeemable zombie trove cannot be made redeemable after receiving a redistribution: before making it redeemable, minting of new debt shares would occur, which reverts because of the 1e9 (*MAX_BATCH_SHARES_RATIO*) limit. So the zombie trove must be first kicked from the batch, then made redeemable, then redeemed.

Code corrected:

In Version 3 of the code `kick_from_batch()` has been correctly implemented.

6.7 Shutdown Price May Be Over-Estimated

Design Low Version 2 Code Corrected

CS-UNCAP-009

WBTCPriceFeed stores a `last_good_price` every time `fetch_price_primary` is called, which takes into effect if oracle failure is detected and the branch is shut down. Depending on the last user operation, `last_good_price` may store the redemption price or the market `WBTC/USD` price. If the last redemption price is recorded right before a failure detected, it might be over-estimated (up to 1% compared to the last market `WBTC/USD` price) and would be used as the price for shutdown operations. Consequently, liquidations and urgent redemptions might become less incentivized and delayed due to the collateral price being in favor of the trove owner.

Code corrected:

In Version 4 the WBTCPriceFeed has been redesigned where the `last_good_price` will only track the `wbtc_price`. Hence the shutdown price will not be over-estimated even it is recorded during a redemption.

6.8 Dust May Be Locked in Collateral Wrapper

Design Low Version 1 Code Corrected

CS-UNCAP-008

Collateral wrapper can wrap a collateral below 18 decimals. Users can wrap the underlying tokens with `wrap()`, which charges the underlying amount and mints an amount upscaled to 18 decimals. Users can unwrap the token with `unwrap()`, which burns the input amount and credits the unscaled amount back to the user:

```
let unscaled_amount = amount / self.scaling_factor.read();
```

Consequently, any amount below `scaling_factor` cannot be unwrapped, but will still be burned leading to lost funds in the contract.

Code corrected:

The issue has been addressed by only burning the amount of wrapped collateral that is actually redeemed. Therefore, the dust balance is not lost but kept in the user's balance.

6.9 Inconsistent Cap Check

Design Low Version 1 Code Corrected

CS-UNCAP-007

The Debt Cap is designed to limit the maximum debt that can be generated on individual branches to restrict the risk that a single asset poses on USDU peg. However, currently the cap check is inconsistent in the codebase. The cap check sometimes accounts for fees & interest and sometimes not. cap check does not account for pending fees & interest in `open_trove()` and `open_trove_and_join_interest_batch_manager()`, but it accounts for pending fees and interest in `_adjust_trove()`

Code corrected:

Functions `open_trove()`, `open_trove_and_join_interest_batch_manager()`, and `_adjust_trove()` in Version 3 all check that the cap is not exceeded without accounting for upfront fee and interest.

6.10 Gas Optimization

Informational Version 1 Code Corrected

CS-UNCAP-014

Inefficient Storage Updates: An inefficient pattern is used when updating structs in storage, where the whole structure is first read, then modified in memory, and then written back to storage entirety. For

example at line 250-254 of BatchManager, the whole batch is read, only 2 of the fields are updated in memory, and then the whole batch is written back to storage, leading to many unnecessary storage writes. The struct fields could be written individually, saving all the reads and most of the writes.

```
let mut batch = self.batches.entry(batch_address).read();
batch.debt = batch_debt + debt_increase;
batch.total_debt_shares = current_batch_debt_shares + batch_debt_shares_delta;
self.batches.entry(batch_address).write(batch);
```

Code corrected:

The inefficient pattern has been replaced with the more efficient setting of individual fields. However, the StoragePath is computed for every field, while it would be more efficient to reuse the already computed StoragePath.

6.11 Missing Getters for `deposit_snapshots`

Informational | Version 1 | Code Corrected

CS-UNCAP-013

The state variable `deposit_snapshots` stores important checkpoints of the user deposits for the yield computation. However, there is no getter function to query the snapshot.

Code corrected:

StabilityPool in Version 3 exposes `get_deposit_snapshots()`.

6.12 Redundant Code

Informational | Version 1 | Code Corrected

CS-UNCAP-015

Unnecessary Type Casting: In Cairo, `try_into().unwrap()` pattern is often used to perform type casts, however, it is unnecessary in many locations since it is casting into the same type, for example in `set_batch_manager_annual_interest_rate()` of BorrowerOperations, variable `last_interest_rate_adj_time` is already of type `u256`:

```
self
    .require_batch_interest_rate_change_period_passed(
        msg_sender, batch.last_interest_rate_adj_time.try_into().unwrap(),
    );
```

Similar unnecessary type casting also appears at line 313 of BatchManager and lines 475, 522, 834, 887, 892 of TroveManager.

Unused Code: There is some unused or duplicate code in the codebase, for instance:

- In contract USDU, `require_caller_is_stability_pool()` is unused
- In StabilityPool, events `TroveManagerAddressChanged` and `USDUTokenAddressChanged` are unused.
- In TroveManager, `owner_to_positions` mapping is unused.

- Function `get_interest_period()` is duplicated in BatchManager and TroveManager.
- Function `calc_interest()` is duplicated in BorrowerOperations, BatchManager, and UncapBase.
- Variables `liquidation_penalty_sp` and `liquidation_penalty_redistribution` are replicated in both AddressRegistry and TroveManager. This could become an issue if during an upgrade the values are changed in only one of the contracts
- Variable `BCR_ALL` is defined however it is never used in the contracts.

Code corrected:

In Version 3 of the code the following points have been addressed:

- The unnecessary type castings have been removed.
- `require_caller_is_stability_pool()` has been removed from USDU.
- `owner_to_positions` has been removed.
- BatchManager does not implement `get_interest_period()` anymore, but retrieves it from TroveManager.
- `calc_interest()` has been factored out of BorrowerOperations, BatchManager, UncapBase, into the `shared.cairo` library.
- In TroveManager, `liquidation_penalty_sp` and `liquidation_penalty_redistribution` are now retrieved from AddressRegistry.

In Version 4 the `BCR_ALL` variable has been removed.

6.13 Stale Cairo Compiler Configuration

Informational Version 1 Code Corrected

CS-UNCAP-016

`Scarb.toml` references Cairo version `2.11.4`, however Uncap Finance does not compile with this compiler version.

Code corrected:

In Version 3 the Cairo compiler version is updated to `2.12.0`.

6.14 Typos and Misnamed Functions & Variables

Informational Version 1 Code Corrected

CS-UNCAP-005

Typos:

In BorrowerOperations:

// Batch managers set the interest rate for every Trove in the batch. The interest rate **it** the same for all Troves in the batch.

In LiquidationManager, a variable is called `coll_to_send_so_SP` instead of `coll_to_send_to_SP`. And there is a typo of **Differencen**.

In CollateralRegistry, a comment mentions `Soft Collateral Ratio` instead of `Shutdown Collateral Ratio`

Misnamed Functions & Variables:

- In RedemptionManager, event arg `strk_fee` of RedemptionFeePaidToTrove is misnamed, as it represents the collateral fee.
- In TroveManager `get_total_coll_and_debt()` is misnamed, as it returns the gain snapshots per unit of stake and not totals.
- In TroveManager `update_trove_id_coll()` updates the collateral of a trove, not trove id.
- In TroveManager `get_trove()` and `get_reward_snapshots()` have a parameter called index, but it is a trove id and not an index (troves have index in the global trove array but this is not it).
- In BorrowerOperations function `open_position_and_join_position()` introduced in Version 3 seems to be misnamed.

Incorrect comments:

BorrowerOperations contains a comment saying:

```
// Check if sponsor has enough approval on gas pool AND if it has enough STRK to pay
```

It is incorrect, as the approval in on BorrowerOperations, not GasPool.

i257.cairo contains the following comments:

```
// i257 represents a 129-bit integer.
```

It is incorrect, as i257 represents a 257 bits integer.

Corrected:

- The comment regarding the interest rate in BorrowerOperations has been fixed.
- The typos in LiquidationManager have been addressed.
- Misnamed `function get_total_coll_and_debt()` has been removed.
- `update_trove_id_coll()` has been renamed to `update_trove_coll()`.
- In `get_trove()` and `get_reward_snapshots()`, parameter `index` has been rename to `trove_id`.
- "Soft Collateral Ratio" in CollateralRegistry has been fixed to "Shutdown".
- The BorrowerOperations comment has been corrected to `approval on BorrowerOperations`.
- i257 comment about `129-bit integers` has been changed to `257-bit integers`.
- Misnamed `open_position_and_join_position()` has been changed to `open_position_and_join_interest_batch_manager()`.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Shutdown Might Not Be Triggered if BTC/USD Oracle Is Down

Informational **Version 2**

CS-UNCAP-003

WBTCPriceFeed is a composite price feed where both `WBTC/USD` and `BTC/USD` price are queried and checked for liveness to determine the redemption price. For other operations, only the `WBTC/USD` price is queried.

Consequently, for non-redemption operations, if the `BTC/USD` oracle is down, the oracle failure will not be detected and a shutdown will not be triggered.

In **Version 4**, WBTCPriceFeed is redesigned where Chainlink oracle works as a fallback if Pragma failure is detected. Note that shutdown will not be triggered in case only Chainlink failure is detected.

7.2 Max Number of Collaterals Is Too High

Informational **Version 1**

CS-UNCAP-004

The maximum number of collaterals is set to 255 in the CollateralRegistry:

```
assert(num_collaterals < Bounded::<u8>::MAX, 'CR: Max collaterals reached');
```

However, a single branch redemption costs around 17M L2 gas, and a transaction on Starknet is capped to 1B L2 gas (as of September 2025). Since every redemption goes through every branches, the system cannot have more than ~1B/17M ~= 50 collaterals.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Collateralization Rates Consideration

Note Version 1

In theory, CCR (Critical Collateralization Ratio) and MCR (Minimum Collateralization Ratio) should be set higher than the SCR (Shutdown Collateralization Ratio), otherwise, upon the deployment of a new branch, one can intentionally trigger a branch shutdown with the following steps:

1. Create the only trove in this branch, which is slightly above MCR.
2. In a following block, due to interest accrual on this trove, the TCR falls below SCR and a shutdown can be triggered.

8.2 Known Issues of Liquity V2

Note Version 1

Since Uncap Finance is a rewrite of Liquity V2 in Cairo, it shares similar known caveats and issues as listed in Bold README and Bold audit report. Following are some selected highlights:

Unredeemable Troves Can Pay Minimum Interest Rate

Zombie troves may be created with debt redistribution during a liquidation. This debt will not be redeemable and those troves can still pay only the minimum interest rate, as long their debt stays below MIN_DEBT since they are unredeemable. Note intentionally creating troves like this is difficult, as it requires liquidations to take place when the stability pool is completely empty, which should only happen in extreme circumstances.

Small Redemptions Do Not Increase Base Rate

The function `CollateralRegistry.get_updated_base_rate_from_redemption` rounds down the new base rate from the share of USDU tokens that are redeemed.

```
let new_base_rate = decayed_base_rate + redeemed_usdu_fraction / REDEMPTION_BETA;
```

As the redeemed_usdu_fraction is rounded down, splitting a redemption into multiple smaller ones can reduce the fee paid. The most extreme case are redemptions with `redeemAmount < total_usdu_supply / 1e18`, which will have their fraction rounded to zero, meaning they will not increase the base rate at all. However, multiple redemptions will incur higher gas costs.

Add Manager Can Increase Stake

The `add_manager` can add collateral to a trove (or anyone can do it if set to 0). However, increasing the collateral of a trove also increases its stake. In extreme scenarios where redistributions give more debt to active positions than collateral, an attacker could send collateral to another trove owner to increase their share in the bad redistributions. Users need to set a trusted `add_manager` if they want to prevent this scenario.

Sending NFTs Does Not Reset Delegation

Troves are transferable NFTs which could be sold on a marketplace. Users purchasing a trove should be aware that the delegation of the trove is not reset when the trove is transferred. This means the seller could set themselves as the remove manager and receiver before the transfer is executed (potentially setting it at the last second, frontrunning the purchase), then remove collateral from the trove after the transfer is completed. As a result, purchasers of troves should be cautious and ensure that the purchase transaction includes a reset of the delegation.