Granite Audit Report

Version 1.1

Conducted by:
Alin Mihai Barbatei (ABA)

August 12, 2024

Table of Contents

1 Introduction

1.1 About ABA

ABA, or Alin Mihai Barbatei, is an established independent security researcher with deep expertise in blockchain security. With a background in traditional information security and competitive hacking, ABA has a proven track record of discovering hidden vulnerabilities. He has extensive experience in securing both EVM (Ethereum Virtual Machine) compatible blockchain projects and Bitcoin L2, Stacks projects.

Having conducted several solo and collaborative smart contract security reviews, ABA consistently strives to provide top-quality security auditing services. His dedication to the field is evident in the top-notch, high-quality, comprehensive smart contract auditing services he offers.

To learn more about his services, visit ABA's website abarbatei.xyz. You can also view his audit portfolio here. For audit bookings and security review inquiries, you can reach out to ABA on Telegram, Twitter (X) or WarpCast:

`https://t.me/abarbatei`

`https://x.com/abarbatei`

`https://warpcast.com/abarbatei.eth`

1.2 About Granite

Granite is a DeFi lending market that offers overcollateralized loans on SIP-10 tokens managed and operated by immutable smart contracts. Granite was created and is managed by Trust Machines, a leading team of engineers, builders and researchers within the Stacks Bitcoin L2 ecosystem.

The protocol is designed to work with three user groups:

- **Lenders:** Individuals or entities looking to earn interest on their crypto assets
- **Borrowers:** Users in need of liquidity that are unwilling to sell their crypto holdings
- **Liquidators:** earn a fee by closing positions

Besides these three, there are also two supporting roles operating on Granite:

- **Governance:** the DAO, which handles critical system operations, a trusted entity
- **Guardians:** special users or bots whitelisted by the DAO that can pause the system in case of extreme emergencies

The protocol consists of a core immutable part containing the state variables, the **`state`** contract, and independent modules that can be changed by governance if needed, to upgrade specific functionalities of the system.

The modules contain the core logic of Granite and are split into **Lender**, **Borrower**, **Liquidation**, **Interest Rate**, and **Oracle** modules, each with their respective functionality.

1.3 Issues Risk Classification

The current report contains issues, or findings, that impact the protocol. Depending on the likelihood of the issue appearing and its impact (damage), an issue is in one of four risk categories or severities: *Critical*, *High*, *Medium*, *Low* or *Informational*.

The following table show an overview of how likelihood and impact determines the severity of an issue.

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behavior that's not so critical.

Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

Informational findings

Informational findings encompass recommendations to enhance code style, operations alignment with industry best practices, gas optimizations, adherence to documentation, standards, and overall contract design.

Informational findings typically have minimal impact on code functionality or security risk.

Evaluating all vulnerability types, including informational ones, is crucial for a comprehensive security audit to ensure robustness and reliability.

2 Executive Summary

2.1 Overview

Project Name	Granite
Codebase	https://github.com/Trust-Machines/granite
Operating platform	Stacks
Programming language	Clarity
Initial commit	`c1bd10baab7639b39584f937f8cb66d3486d9a75`
Remediation commit	`566deb47b5b3296ab5dc208c1ac7ab4d5325c7fb`
Timeline	From 25.06.2024 to 12.07.2024 (18 days)
Audit methodology	Static analysis and manual review

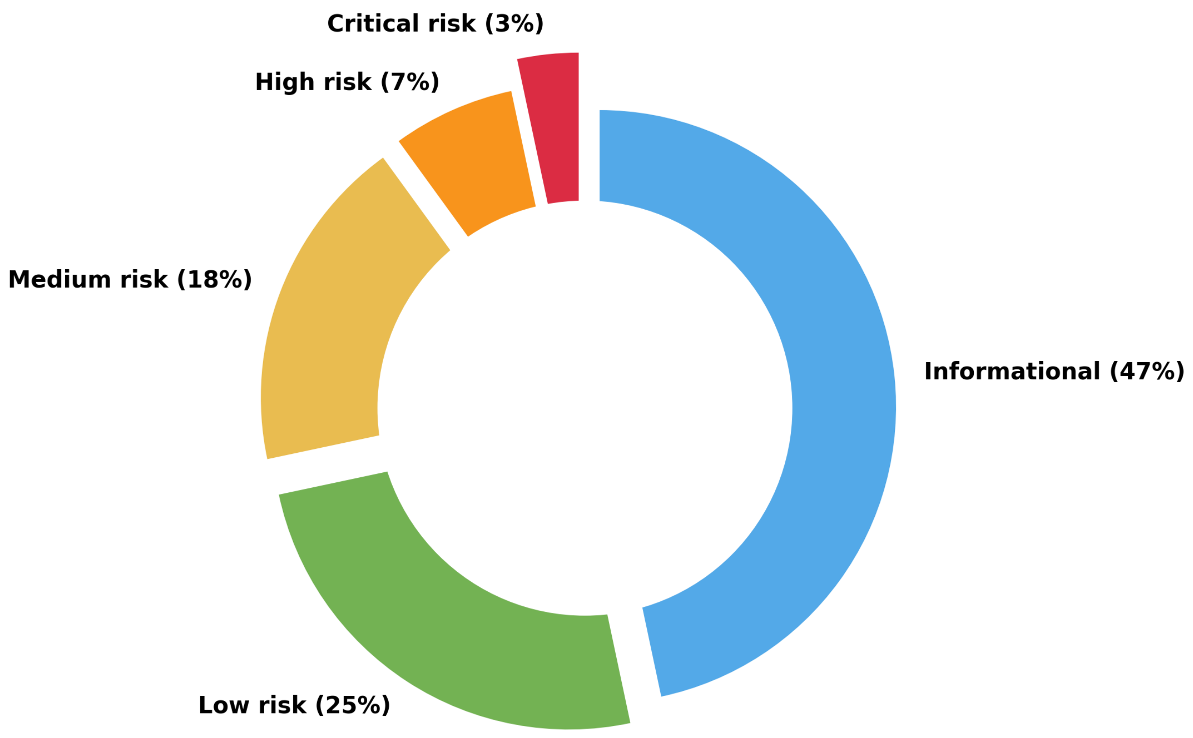
2.2 Audit Scope

Files and folders in scope

```
- contracts/state.clar
- contracts/borrower.clar
- contracts/liquidator.clar
- contracts/liquidity-provider.clar
- contracts/governance.clar
- contracts/meta-governance.clar
- contracts/modules/pyth-oracle.clar
- contracts/modules/math.clar
- contracts/modules/linear-kinked-ir.clar
- contracts/modules/bundler.clar
- contracts/traits/trait-sip-010.clar
```

2.3 Summary of Findings

Severity	Total Found	Resolved	Partially Resolved	Acknowledged
Critical risk	2	2	0	0
High risk	4	4	0	0
Medium risk	11	9	0	2
Low risk	15	15	0	0
Informational	28	25	0	3



2.4 Findings & Resolutions

ID	Title	Severity	Status
C-01	Protocol funds are blocked in the multisig	Critical	Resolved
C-02	Bad debt cannot be repaid	Critical	Resolved
H-01	Current pyth-oracle implementation is incorrect	High	Resolved

ID	Title	Severity	Status
H-02	Unnecessary scaling on debt and lending conversions block contract operations at high TVL	High	Resolved
H-03	Several liquidation flows revert or result in less collateral reward for liquidator	High	Resolved
H-04	Differing decimal tokens are incorrectly handled	High	Resolved
M-01	Repaying a loan or adding collateral should not be blocked if liquidations are allowed	Medium	Resolved
M-02	Unblocking top-ups, repays and liquidations too close to each other can prevent users from saving their positions	Medium	Resolved
M-03	Setting a cap on deposited collateral risks users becoming liquidatable without being able to save their position	Medium	Resolved
M-04	Collateral support cannot be completely dropped	Medium	Resolved
M-05	Reducing collateral liquidation LTV ratio may instantly liquidate users	Medium	Acknowledged
M-06	Lack of slippage on liquidations	Medium	Resolved
M-07	Rounding in liquidations is against the protocol	Medium	Resolved
M-08	Governance proposals never expire and can't be rejected	Medium	Resolved
M-09	Governance multisig can be corrupted by adding existing or removing inexistent members	Medium	Resolved
M-10	account-health does not take into consideration pending interest	Medium	Acknowledged
M-11	Anyone can liquidate a position with bad debt	Medium	Resolved
L-01	Missing getter for deposit and withdrawal status	Low	Resolved
L-02	Governance multisig member count should not be allowed to reach 0	Low	Resolved
L-03	Interest accrual cannot be suspended	Low	Resolved
L-04	Protocol upgradeability cannot be freezed	Low	Resolved
L-05	Collateral tokens with a maximum LTV of 0 should not be accepted as collateral	Low	Resolved
L-06	Protocol reserve percentage can be set to over 100%	Low	Resolved
L-07	Small loans may be unprofitable to liquidate	Low	Resolved

ID	Title	Severity	Status
L-08	Liquidations are blocked if collateral price reaches 0	Low	Resolved
L-09	All liquidity related calls should go through the bundler contract to have up-to-date prices	Low	Resolved
L-10	Stacks reorgs may affect governance proposals	Low	Resolved
L-11	Add extra protection against self-liquidations	Low	Resolved
L-12	There are no sanity checks when updating interest accrual settings	Low	Resolved
L-13	Avoid using tx-sender for sensitive operations	Low	Resolved
L-14	Collateral discount and liquidation LTV incompatible values can crash liquidations	Low	Resolved
L-15	Liquidations do not prioritize the lowest LTV asset	Low	Resolved
I-01	Reuse SUCCESS constant	Informational	Resolved
I-02	Remove unused constants	Informational	Resolved
I-03	borrowing::borrow can be simplified	Informational	Resolved
I-04	state::update-borrow-state can be simplified	Informational	Resolved
I-05	state::update-liquidate-collateral-state can be simplified	Informational	Resolved
I-06	governance::execute-state-reserve-action can be simplified	Informational	Resolved
I-07	Use errors instead of panicking with slight codebase simplifications	Informational	Resolved
I-08	governance::execute-update-guardian can be improved	Informational	Resolved
I-09	is-contract-deployer can be inlined	Informational	Resolved
I-10	Removing a governance member does not invalidate his casted votes	Informational	Acknowledged
I-11	There is no reserve_admin role	Informational	Resolved
I-12	state::toggle-upgrades function name is misleading	Informational	Resolved
I-13	Remove debug remnants	Informational	Resolved
I-14	Fully liquidating a collateral position does not remove the user collateral principal	Informational	Acknowledged
I-15	Apply the extended Clarity Style Guide	Informational	Resolved
I-16	linear-kinked-ir::accrue-interest can be simplified	Informational	Resolved

ID	Title	Severity	Status
I-17	state::increase-total-assets must never be executed permissionless	Informational	Resolved
I-18	Typographical errors	Informational	Resolved
I-19	English dialect inconsistencies	Informational	Resolved
I-20	Add descriptive error code for when no debt is present but health check is done	Informational	Resolved
I-21	liquidator::liquidate-collateral can be simplified in conjunction with liquidator::account-health	Informational	Resolved
I-22	Liquidation discount factor may be counterproductive in some conditions	Informational	Resolved
I-23	linear-kinked-ir::utilization-calc can be simplified	Informational	Resolved
I-24	Remove redundant comments	Informational	Acknowledged
I-25	Inconsistent contract deployer handling throughout the codebase	Informational	Resolved
I-26	Redundant safe-div call in liquidator::calc-collateral-to-give	Informational	Resolved
I-27	Check fees and limitations after pyth oracle deployment	Informational	Resolved
I-28	Consider applying an anchoring pattern to the scaling-factor constant	Informational	Resolved

3 Findings

3.1 Critical Severity Findings

[C-01] Protocol funds are blocked in the multisig

Severity: ***Critical risk*** *(Resolved)* *[PoC]*

Context: ***governance.clar***:*276-278* ***state.clar***:*395,402*

Description

Governance manages the protocol reserve, a cushion of liquidity made by a fee cut on the interest rate paid by borrowers and acting as a last resort to facilitate withdrawals if the protocol is empty as well as absorb bad liquidations. The protocol reserve is also the primary source of revenue for the protocol team, from where all expenses are paid.

In the `governance` contract through the `ACTION_DEPOSIT_TO_RESERVE` and `ACTION_WITHDRAW_FROM_RESERVE` actions, the protocol team can both deposit extra funds into the Granite market reserve fund and withdraw from it.

However, the withdrawal operations is flawed because it transfers the funds from the `state` contract to the `governance` contract, instead of the caller or a designated receiver.

```
(define-private (execute-state-reserve-action (proposal-id uint) (action uint))
  ;; ... code ...
  (if (is-eq action ACTION_WITHDRAW_FROM_RESERVE)
    (contract-call? .state withdraw-from-reserve amount)
    ERR-INVALID-ACTION
  )
  ;; ... code ...
```

The `governance` contract does not have any way of transferring the tokens out of itself.

If we look at the situation from a different angle, the root issue may be that the `state::withdraw-from-reserve` transfers the tokens to the contract caller (`contract-caller`) instead of the transaction sender (`tx-sender`)

```
(define-public (withdraw-from-reserve (amount uint))
  (let (
      ;; ... code ...
      (recipient contract-caller)
    )
    ;; ... code ...
    (as-contract (try! (contract-call? .mock-usdc transfer amount (as-contract
        tx-sender) recipient none)))
    ;; ... code ...
)
```

With the current implementation, the team can never withdraw any funds from the protocol reserves.

Recommendation

Modify the `governance` contract to support withdrawing funds. Can be done via a separate action.

Doing it like so is desirable as withdrawing assets to the last principal that approved the proposal, an alternative, should not be done. Funds should be sent to a treasury principal.

Also, an advantage in supporting withdraws from the multisig, is that any tokens that end up in the `governance` contract by mistake can be rescued.

Resolution: Resolved, the recommended fix was implemented in **PR#121**.

[C-02] Bad debt cannot be repaid

Severity: ***Critical risk*** *(Resolved)* *[PoC]*

Context: ***liquidator.clar***

Description

In a borrowing and lending protocol, if a borrower is unable to cover his debt with all his available collateral, then he has bad debt and implicitly the protocol has bad debt.

Bad debt results in lenders being unable to withdraw their deposited market tokens, making any protocol with outstanding bad debt undesirable for market participants.

In Granite, any liquidator can create bad debt, contrary to what the documentation indicates. Meaning that market participants can call `liquidator::liquidate-collateral` leaving the liquidated with no collateral but debt.

This type of behavior is normal with liquidations as bad debt is naturally occurring within any economical market system. Over time, mechanism have been constructed to help deal with bad debt that accumulates within a system. Mechanism such as a 3rd party repaying the debt or socializing the debt.

However, in Granite there is no such mechanism implemented. This is a critical issue as bad debt will accumulate since nobody can repay it as it continues to accumulate interest. Lenders will not be able to withdraw market token and further market participants will not interact with the protocol.

There are two largely known approaches in resolving bad debt:

1. Have someone else repay the bad debt, not the borrower, effectively making a donation to the system
2. Socialize debt, where on liquidations, the bad debt is absorbed by all the market lenders as an equal share

For the first approach, in practice the protocol team or governance would need to manually repay all the positions left with bad debt as no economical agent would willingly lose funds without any incentive. Governance has the incentive of making the protocol as attractive as possible for market participants.

Socializing debt, the second approach, is a popular bad-debt elimination mechanism and can coexist with donation repayments. Governance can even have a special option to activate or deactivate it, a commonly seen pattern.

To socialize debt, after all the borrower's collateral has was extracted through liquidations, have the remaining debt be taken out of the existing total market assets amount. A mechanism would be required to determine if a user has any leftover collateral and if not, execute the debt socialization.

Known limitations with debt socializing

There are a number of known issues or limitations associated with bad debt socializing. A discussion regarding them and their impact:

- The borrower (or an ill intended liquidator) can liquidate the position while leaving dust amounts of collateral. Thus, other liquidators would not be incentivize to liquidate the position, leading to the accumulation of bad debt. Although seemingly impactful, this issue has not been observed to have such a large impact over the years. For example, Morpho, a leading borrowing and lending platform, chose to accept this potential issue in its (highly) secure vaults (see H-01 Bad Debt socialization Can Be Maliciously Skipped)
- An opportunistic lender may observe an upcoming large liquidation that has bad debt and withdraws their deposit beforehand. After the debt socialization, they would reenter the system. This has also been observed over the years to not manifest to an extent that it impacts the markets. A compelling explanation from a game-theory point of view is presented by Euler, in their V2 implementation of bad debt socialization
- Small positions that are unprofitable for liquidators to close will accumulate naturally over the years. Some protocols, like MakerDAO have *introduced a dust factor that forbids loans with less than a threshold amount of collateral estimated to make the loan profitable* (the quoted information is from @alcueca's article on liquidations). Adding a minimum borrow limit in practice is hard to maintain while the benefits have not clearly been shown over time.

Recommendation

Extend the `repay` functionality to allow repaying someone else's debt.

Also implement bad deb socialization. In the `state::update-liquidate-collateral-state` function, verify if a user's balance is 0 after removing the `collateral-to-give`. If so, remove the collateral principal from the `user-collaterals` map and also from the `positions` map.

After the removal, if user does not have any other collateral principal available, execute debt socializing.

Implement a flag to turn on or off bad deb socialization with a corresponding governance action to change it. Have bad debt socialization on by default.

Until actual market experience and conditions have shown otherwise, postpone:

- implementing any minimum loan size
- adding restrictions or limitations on lenders withdrawing their tokens

Any dust positions that naturally occur over the years that are not liquidated due to them not being profitable to users, manually liquidated them at a point. By having debt socialization implemented by default, this mechanism will result in a relatively slow rate of increase.

For a Clarity based smart contract, which has the extra burden of needing to be as simple as possible due to current technological constraints, over-engineered mitigation mechanism that are not clearly beneficial or even implemented in already existing prominent lending protocols, do not have sense to be added.

Mentioned reading resources on liquidations and debt socialization:

- Morpho uses bad debt socialization
- Euler built their v2 on the same premise as Morpho and socialize debt
- Liquidations in Decentralized Finance: A Comprehensive Review by @alcueca

Resolution: Resolved, the recommended fix was implemented in **PR#154**.

3.2 High Severity Findings

[H-01] Current pyth-oracle implementation is incorrect

Severity: ***High risk*** *(Resolved)* *[PoC]*

Context: ***pyth-oracle.clar****:21-23,30-47,49-56,94-111*

Description

For determining token value, Granite will use the pyth oracle network. Since Granite will be deployed only after Nakamoto, there are still known changes that are to be implemented at that point, such as the price staleness check in the `pyth-oracle` contract, or hardcoding the correct mainnet versions of pyth contracts instead of the current testnet versions.

However, there are other changes and issues that need to be resolved:

1. issue: Within the `pyth-oracle` contract, price feeds are incorrectly mapped to token tickers instead of principals (token addresses).

```
;; NOTE: assuming collateral tickers are less than 10 chars in length
;; price feeds can be found in <https://pyth.network/developers/price-feed-ids>
(define-map price-feeds (string-ascii 10) (buff 32))
```

2. issue: There is no governance action which would call `update-price-feed-id` implemented in the `governance` contract, but the function can only be called by governance.

3. improvement: The caller check from `update-price-feed-id` should validate using `contract-caller` instead of `tx-sender`, similar to how the `state::is-governance` function validates.

```
(asserts! (is-eq (contract-call? .state get-governance) tx-sender)
    ERR-NOT-AUTHORIZED)
```

This is just as optimization as the future governance action can simply use `as-contract` to ensure the check passes.

4. issue: The `(define-public (read-price (ticker (string-ascii 10))) ...` function from the `pyth-oracle` contract must also be changed because within the Granite codebase, tokens are saved as principals, not ticker strings.

5. issue: The `convert-res` function which is responsible for scaling or down-scaling decimals from the returned pyth response to a desired `resolution-digits` value is incorrectly implemented. In some cases reverts, in others returns an incorrect value.

It is incorrect because it calculates the difference between decimals `diff` by subtracting the desired decimals from the non-absolute value of the pyth returned exponential, which is a negative value `(diff (abs (- expo resolution-digits)))`. As such, the diff is larger and ultimately results in a trimmed down, incorrect, price.

It also incorrectly treats all cases where the pyth exponential is larger than 0 and the case where the exponential is 0 but the desired decimals are not 0.

The function can be reimplemented as:

```
(define-private (convert-res (price int) (expo int) (resolution-digits int))
  (if (>= expo 0)
    (* price (pow 10 (+ expo resolution-digits)))
    (let
      ( (diff (- resolution-digits (abs expo))) )
      (if (is-eq diff 0)
        price
        (if (> diff 0) (* price (pow 10 diff)) (/ price (pow 10 (abs diff))))
      )
    )
))
```

The adjacent POC includes the tests for the function.

Recommendation

Implement the mentioned changes and fixes.

Resolution: Resolved, the recommended fix was implemented in **PR#128**.

[H-02] Unnecessary scaling on debt and lending conversions block contract operations at high TVL

Severity: ***High risk*** *(Resolved) [PoC]*

Context: ***math.clar****:24-54* ***liquidity-provider.clar****:48-77*

Description

Issue when borrowing assets

In the `math` contract, the `convert-to-debt-shares` and `convert-to-debt-assets` functions are used, as per their name, to convert debt shares and borrowed market tokens (assets) one to each other.

Both of these functions apply a scaling factor (1e8) with the purpose of reducing rounding errors. However, the scaling does not help, it increases code complexity and also introduces a critical overflow error.

First, consider the following logical equivalence for each operation within the functions to show how the scaling negates itself.

- for: `math::convert-to-debt-shares`

Operation	Equivalence
(scaled-total-shares (* (get total-debt-shares debt-params) scaling-factor))	total-debt-shares * scaling-factor
(scaled-num (* assets scaled-total-shares))	assets * total-debt-shares * scaling-factor

Operation	Equivalence
(scaled-den (* (get open-interest debt-params) scaling-factor))	open-interest * scaling-factor
(divide round-up scaled-num scaled-den)	(assets * total-debt-shares * scaling-factor) / (open-interest * scaling-factor)

We can see that mathematically the last operation can be rewritten as `assets * total-debt-shares / open-interest` without loosing any more precision that with scaling applied.

- for: `math::convert-to-debt-assets`

Operation	Equivalence
(scaled-den (* total-shares scaling-factor))	total-debt-shares * scaling-factor
(scaled-num (* (* (get open-interest debt-params) scaling-factor) shares))	open-interest * scaling-factor * shares
(divide round-up scaled-num scaled-den)	(open-interest * scaling-factor * shares) / (total-debt-shares * scaling-factor)

Like in the previous case, we can see that mathematically the last operation can be rewritten as `open-interest * shares / total-debt-shares` without loosing any more precision than with scaling applied.

Coming back to the overflow issue, in both functions, at one point there is a multiplication of three variables.

In the `convert-to-debt-shares` function, the multiplication `assets * total-debt-shares * scaling-factor` overflows when the resulting value surpasses the maximum of `uint128`.

The maximum of `uint128` can be approximated to `10e39`, scaling-factor is `1e8`, and `total-debt-shares` is also has `1e8` decimals.

As such, this operation reverts when the multiplication result is greater then `1e15`.

To give an example, for a system with 500 million debt shares. This amount of debt shares, at minimum, would require 500 million USDC being borrowed in one transaction, which is not realistic. Multiple small borrows would need slightly more capital to reach this share debt amount due to how shares are calculated. Continuing with this example, at 500 million debt shares any borrow over $680,565 would revert (`max(uint128) / (scaling-factor * total-debt-shares)`).

While exemplified values seem unrealistic, the more debt the system takes, the less maximum borrowing is allowed. Also considering the Bitcoin has the highest market in existence, over time, this issue will prove to be a blocker.

The `convert-to-debt-assets` function also has the same issue with the multiplication `open-interest * scaling-factor * shares`.

Issue when lending assets

As with borrowing, the same scaling redundancy and issue is present when depositing (lending) or withdrawing market tokens from Granite. Specifically the functions `convert-to-shares` and `convert-to-assets` from the `liquidity-provider` contract both errantly scale.

- for: `liquidity-provider::convert-to-shares`

Operation	Equivalence
(scaled-total-shares (* (get total-shares asset-params) (get scaling-factor asset-params)))	total-shares * scaling-factor
(scaled-num (* assets scaled-total-shares))	assets * total-shares * scaling-factor
(scaled-den (* (get total-assets asset-params) (get scaling-factor asset-params)))	total-assets * scaling-factor
(contract-call? .math divide round-up scaled-num scaled-den)	(assets * total-shares * scaling-factor) / (total-assets * scaling-factor)

where `(assets * total-shares * scaling-factor) / (total-assets * scaling-factor)` is equivalent to `total-shares * assets / total-assets` without precision loss.

- for `liquidity-provider::convert-to-assets`

Operation	Equivalence
(scaled-den (* (get total-shares asset-params) (get scaling-factor asset-params)))	total-shares * scaling-factor
(scaled-num (* (* (get total-assets asset-params) (get scaling-factor asset-params)) shares))	total-assets * scaling-factor * shares
(contract-call? .math divide round-up scaled-num scaled-den)	(total-assets * scaling-factor * shares) / (total-shares * scaling-factor)

where `(total-assets * scaling-factor * shares) / (total-shares * scaling-factor)` is equivalent to `total-assets * shares / total-shares` without precision loss.

In this situation, users will not be able to deposit more market tokens over time.

Other redundancies

All the four mentioned functions also present undescriptive naming for the intermediary variables `scaled-den` and `scaled-num` while also having some code duplication. After scaling is removed, the intermediary variables are also not needed.

Recommendation

Rewrite the functions without scaling and optimizing the code as much as possible.

Resolution: Resolved, the recommended fix was implemented in **PR#129**.

[H-03] Several liquidation flows revert or result in less collateral reward for liquidator

> **Severity:** ***High risk*** *(Resolved)* *[PoC]*
>
> **Context:** ***liquidator.clar****:113-154*

Description

When a position is liquidated, the liquidator chooses the collateral token for the liquidation reward and how much of the user's debt is he willing to pay. The `liquidator::liquidate` function determines the amount of collateral tokens the liquidator will receive, with the liquidation discount included, and how much he is actually required to pay.

The actual amount to repay, `repay-amount`, can be lower than the original amount the liquidator passed because it is intentionally capped to avoid:

- repaying more debt resulting in a position health above 1. At a health value of 1, the position is considered borderline healthy and by protocol design, liquidations are not allowed to bring a position past this point
- repaying more debt that the value of the discounted collateral would bring. This case would result in liquidators losing collateral on liquidations

During liquidation, there are several execution flows where `liquidate` determines invalid repay and collateral amounts, resulting in liquidation being reverted instead.

Case 1: Liquidations revert when over repaying for multi-collateral loans

In a situation where a borrower has multiple collateral tokens, if the liquidator passes a repay amount greater than the maximum allowed repaid, then the liquidation will revert instead of being capped to the maximum allowed limit.

In the `liquidate` function, `liquidator-repay-amount` is how much the liquidator at maximum will pay and `total-repay-amount` is how much can precisely be paid in order to

```
(repay-amount (if (< total-repay-amount liquidator-repay-amount)
    total-repay-amount
    ;; ... else branch code ...
```

As we can see from the code, if the total amount the liquidator is willing to pay is more than the maximum allowed, then the `total-repay-amount` is assigned to the `repay-amount`. The repay amount is the final amount the liquidator will pay for the given collateral.

The exact collateral to be received is calculated in the same function and is equivalent to the same value as the repaid amount plus an extra percentage, representing the discount equivalent.

```
;; calculate collateral to give based on repay amount with discount considered
(collateral-to-give (if (is-eq collateral-value repay-amount)
    deposited-collateral-amount
    (try! (calc-collateral-to-give repay-amount liquidation-discount
       collateral-price resolution-factor))
  )
)
```

For a multi-collateral loan, the issue appears because the `repay-amount` is calculated as if paying the entire debt had been possible while giving collateral tokens to the liquidator from only one collateral position.

In a one collateral loan position that would be true, but in a multi-collateral position the resulting `collateral-to-give`, being based on a `repay-amount` which considers only one collateral repay, will become a value that is significantly higher then the actual collateral amount a users has.

In this particular case `repay-amount` will not be equal to the collateral value because logically, the user reached liquidation when all of his multi-collateral positions did not bring him sufficient value to be healthy. Thus the collateral value of one token cannot be suffice to repay the entire amount, resulting in `collateral-to-give` going on the `else` branch and being calculated as `(try! (calc-collateral-to-give repay-amount liquidation-discount collateral-price resolution-factor))`

The else branch of the code, where the collateral amount is increases should never be reached with a repay amount above the collateral value, as in this case.

As a result of the over-inflated repay amount and collateral to give amount, liquidation then will revert in one of two places. It will first revert when transferring collateral out of the `state` contract, for the case when the extra collateral requested does not exist in the contract. If there are enough collateral tokens, it will revert when decreasing the user collateral amount, as it underflows.

Case 2: Liquidations revert when liquidator repay amount value is between then collateral and the collateral value minus the discount

Another issue appears when more than the entire user collateral amount would be taken because of the applied discount. In this case more collateral is taken from the user that he actually has, resulting in a transaction revert.

In the `liquidator::liquidate` function, when the collateral-to-give is calculated, it checks if the calculated liquidator repay amount USD valuation is equal to that of the collateral. If so, it gives to the liquidator the entire user collateral balance.

The issue appears on the else branch, where if it is not maximum, the collateral to give is determined by the `liquidator::calc-collateral-to-give` function that increases the amount to be given by the discount factor.

```
(collateral-to-give (if (is-eq collateral-value repay-amount)
    deposited-collateral-amount
    (try! (calc-collateral-to-give repay-amount liquidation-discount
       collateral-price resolution-factor))
  )
)
```

In this case, if the repay amount is less than the collateral value but the difference is less than the applied discount, the resulting collateral amount will be more than the entire user balance.

Liquidation then will revert in one of two places. It will first revert when transferring collateral out of the `state` contract, for the case when the extra collateral requested does not exist in the contract. If there are enough collateral tokens, it will revert when decreasing the user collateral amount, as it underflows.

Case 3: Liquidator does not get discount when the repay amount value is greater than collateral value but less than total repay amount

Another situation appears when the final repay amount is set to the collateral value itself. In that case, the collateral to give is the entire user deposited amount.

This is incorrect as liquidator should receive a discount even when retrieving the full user deposited collateral.

The issue originates when comparing the liquidator repay amount to the collateral value and then attributing to it if greater.

```
(repay-amount (if (< total-repay-amount liquidator-repay-amount)
    total-repay-amount
    (if (> liquidator-repay-amount collateral-value) collateral-value
        liquidator-repay-amount)
  )
)
```

This leads to the first, true branch, of the collateral-to-give attribution to the entire deposited amount.

```
(collateral-to-give (if (is-eq collateral-value repay-amount)
    deposited-collateral-amount
    ;; else branch code ...
```

The correct comparison is against a reduced collateral value that takes into consideration the discount that will be applied afterwards so that `collateral-to-give` becomes the fully available amount. i.e. comparing `liquidator-repay-amount` with the `adjustedCollateralValue` and if greater, then return the `adjustedCollateralValue` itself:

$\text{adjustedCollateralValue} = \frac{\text{collateralValue}}{1+\text{liqReward}}$

This logic is also indicated in the project documentation regarding the maximum repay amount allowed.

Note: as a POC it is enough to refer to the already existing "Borrower borrows and full liquidation" test, where, on the first liquidation, with a total collateral value of 650 that is liquidated, the liquidator is incorrectly expected to have spent 650 equivalent value (ending balance to be 5000 - 650 = 4350), resulting in the discount loss.

Recommendation

The three issues within the `liquidator::liquidate` function are thus:

1. Liquidations revert when over repaying for multi-collateral loans
2. Liquidations revert when liquidator repay amount value is between then collateral and the collateral value minus the discount

3. Liquidator does not get discount when the repay amount value is greater than collateral value but less than total repay amount

If each issue had been taken in isolation, the fixes would become needlessly overcomplicated and at place would overlap. To ensure an efficient resolution, the simplest way is to rewrite the function so that the liquidator repay amount, capping logic and subsequent collateral-to-give calculation mirror the documentation.

From the project documentation, we can see that:

$maxRepayCalc_x = \frac{debt - \sum(value_i \times liqLTV_i)}{1-(1+liqDiscount_x) \times liqLTV_x}$ and

$maxRepayAllowed_x = \max(\min(maxRepayCalc, \frac{collValue_x}{1+liqReward_x}), 0)$

The `maxRepayCalc_x` in code is correctly calculated and saved in the `total-repay-amount` variable. Also, because the subtraction in it (`(- debt total-collaterals-liquid-value)`) would revert, there is no need for the `max` logic when calculating `maxRepayAllowed_x`.

It is important to have `maxRepayAllowed_x`, as its name suggests, be the highest allowed value for the repay amount. Thus capping liquidator passed repay amount to it.

Improvements to consider:

- `safe-div` is redundant to use when dividing by the resolution factor as it will never be zero, the only case where `safe-div` is useful. Remove all apparitions except when dividing by the collateral price, which can be 0 under extreme situations
- if needed, for better UI integration, an initial check that `debt` is larger then `total-collaterals-liquid-value` can be done, as to not revert with an underflow panic
- the collateral adjusted value would be $\frac{collValue_x}{1+liqReward_x}$
- rounding needs to be taken into consideration when calculating the final repay amount since it depends on the collateral adjusted value in some cases. Rounding up is against the liquidator and rounding down is against the borrower. It is generally better to round against the liquidator as to discourage multiple small liquidations

Example implementation:

```
(define-read-only
  (liquidate
    (debt uint)
    (total-collaterals-liquid-value uint)
    (collateral-value uint)
    (liquidation-discount uint)
    (collateral-liquid-ltv uint)
    (deposited-collateral-amount uint)
    (collateral-price uint)
    (liquidator-repay-amount uint)
    (resolution-factor uint)
  )
  (let (
      (denominator (- resolution-factor (/ (* (+ resolution-factor
          liquidation-discount) collateral-liquid-ltv) resolution-factor)))
      (total-repay-amount (try! (safe-div (* (- debt
          total-collaterals-liquid-value) resolution-factor) denominator)))
      (adjusted-collateral (div-round-up (* collateral-value resolution-factor) (+
          resolution-factor liquidation-discount)))
      (repay-amount (min liquidator-repay-amount (min total-repay-amount
          adjusted-collateral)))
      ;; calculate collateral to give based on repay amount with discount considered
      (collateral-to-give (if (is-eq repay-amount adjusted-collateral)
        deposited-collateral-amount
        (try! (calc-collateral-to-give repay-amount liquidation-discount
            collateral-price resolution-factor))
      ))
    )
    (ok {repay-amount: repay-amount, collateral-to-give: collateral-to-give})
  )
)

(define-private (min (a uint) (b uint)) (if (< a b) a b) )

(define-private (div-round-up (num uint) (den uint))
   (if (> (mod num den) u0) (+ u1 (/ num den)) (/ num den))
)
```

Note:

- the "Borrower borrows and full liquidation" test will fail because it incorrectly considers no liquidation bonus (Issue 3) for the liquidator.
- the above example implementation rounds up the liquidator repay-amount. This behavior benefits the borrower and is compatible with the currently intended logic, as shown by the "should return full collateral" test.

Resolution: Resolved, the recommended fix was implemented in **PR#137**.

[H-04] Differing decimal tokens are incorrectly handled

Severity: ***High risk*** *(Resolved) [PoC]*

Context: ***borrower.clar****:44-46,168-170* ***liquidator.clar****:41,46*

Description

As a borrowing and lending protocol, each Granite deployment will have a market token and users will be able to borrow and lend collateral tokens. Initially Granite will support the `sBTC`, `stSTX`, STX (wrapper) and a USDC equivalent tokens. Besides `stSTX` (Stacked Stacks), none of the other tokens are launched yet.

Granite will require supporting, besides these tokens, any other future token chosen by governance. For that, all tokens must be SPI-10 compliant and non-rebase/fee-on-transfer tokens.

SPI-10 compliant token support, however, does not come with any guarantees regarding the token decimals in question, Granite having the responsibility of correctly handling any decimals a token might have. For example, sBTC will have 8 decimals and stSTX has 6 decimals.

Currently, Granite does not correctly handle any situation where the collateral token decimals differ from the market token decimals.

This causes severe issues on any operation that relies on comparing debt value to collateral value, implicitly any account health is misjudged. This happens because the LTV adjusted collateral value (collateral decimals) will be incorrectly evaluated against the debt value (in market tokens decimals), leading to incorrect liquidations, collateral repayment and borrowing.

Recommendation

Market token and collateral must be brought to the same decimals when compared in any way.

This requires to have at hand all token decimals. Calling the SIP-10 `get-decimals` function where needed is the simples but also the most costly fix. The following approach is advised in stead:

- hardcode the decimals for the market token as a constant in all the contracts where it is needed (`borrower` and `liquidator`). We are already hardcoding the name of token contract (e.g. `.mock-usdc`) itself, no need to call it each time or continuously pass it from the `state` contract. A slightly more costly but a more generic solution is to define the decimal constant as `(define-constant market-decimals (contract-call? <market-token-contract> get-decimals))` in each contract.
- the `collaterals` map should also save the token decimals. Modify the `state::update-collateral-settings` so that it also saves the collateral decimals in the map. The decimals would be here retrieved by a `get-decimals` call
- wherever it is needed, pass the collateral decimals to the point that conversions can be done (shown in the issue **Context** section)
- duplicate the `linear-kinked-ir::to-fixed` function in every contract where conversions are needed (`borrower` and `liquidator`) and use it. We do this in order to avoid multiple calls to the `linear-kinked-ir` contract and save on fees

Resolution: Resolved, the recommended fix was implemented in **PR#168** and **PR#182**.

3.3 Medium Severity Findings

[M-01] Repaying a loan or adding collateral should not be blocked if liquidations are allowed

> **Severity:** *Medium risk* *(Resolved)*
>
> **Context:** ***state.clar****:223-235,279-291,265-277*

Description

Governance has the possibility of blocking specific operations from being executed by the protocol. Liquidating a position, paying back a loan and adding more collateral to a position are operations where each has its own pause flag, respectfully `liquidation-enabled`, `repay-enabled` and `add-collateral-enabled`. There is also the option of fully stopping the entire market via the `pause-market` function from the `state` contract.

If, at any time, repaying or adding collateral is paused but liquidations are not paused, then users are unfairly put in a position where they can be liquidated but cannot intervene to save their collateral.

Recommendation

Allow disabling repays and disabling adding collateral only if liquidations are also disabled. Also, enabling liquidations should not be allowed if replays and adding collateral are not permitted. These changes would also be required to be done in the `pause-market` and `unpause-market` functions of the `state` contract.

Depending on protocol business requirements, pausing and unpausing all three operations at the same time may be implemented. This would be performed as a separate action within the `governance` contract that does: deactivation of liquidation, repaying loans and adding collateral at the same time but in that order. This would be an alternative to `unpause-market` which also pauses other actions.

The checks would be better one in the `governance` contract as to not further increase the `state` contract size and complexity.

Resolution: Resolved, the recommended fix was implemented in **PR#136** and **PR#166**.

[M-02] Unblocking top-ups, repays and liquidations too close to each other can prevent users from saving their positions

> **Severity:** *Medium risk* *(Resolved)*
>
> **Context:** ***state.clar****:223-235,279-291,265-277*

Description

At any given time, due to a variety of reasons, the protocol may block certain user operations. Considering a situation where liquidations, the addition of collateral and repaying debt are blocked or paused.

In this case, the price of the underlying collateral and market tokens can still fluctuate. Because of that, borrower positions may become unhealthy during the system pause.

If all operations are unblocked close to each other or at the same time, via a `unpause-market` call, users will not be able to save their positions from being liquidated.

Recommendation

Implement a grace period in which liquidations, after being enabled, will not be allowed if a specific time internal has not passed since unblocking. Liquidations should not be allowed if topping-up collateral and repaying debt is not enabled (this is mentioned as a separate issue).

This solution would also protect users when the system was paused via a `pause-market` call and unpausing would commence via the `unpause-market` function.

Resolution: Resolved, the recommended fix was implemented in **PR#157**.

[M-03] Setting a cap on deposited collateral risks users becoming liquidatable without being able to save their position

> **Severity:** *Medium risk* *(Resolved)*
>
> **Context:** ***borrower.clar***:*131*

Description

Each Granite market has a cap on the maximum amount users can deposit as collateral. This is done to ensure a gradual TVL growth and prevent excessive capital inflows at the beginning of the protocol life.

However, capping user added collateral results in users being unable to save themselves when nearing liquidation. There should never be a situation where a user can be liquidated but he cannot add more collateral to his position or repay it.

For example, when a market reaches the deposited collateral cap, all existing loans at this point are forced to either repay the debt or be liquidated. By being liquidated, collateral max deposit availability is increased and other positions can now save themselves from liquidation. This also creates malincentives between borrowers.

An extreme case where adding collateral should be blocked, and a cap can help, is if a collateral token has been deemed unsuited for the lending protocol due to market and economical reasons and has its maximum LTV reduced to 0, users can still add it as collateral if the collateral debt cap is not set. But here the cap works as a disable/enable mechanism.

Recommendation

In order to ensure a gradual TVL growth, the natural user action flow must be limited. In a healthy and prosperous market, lenders want to deposit market tokens to get the interest. Also borrowers want to borrow as much as they can while limiting the collateral they deposit. Collateral deposit is already an action which is never naturally overdone, as there is no incentive for users to do so.

Lending market tokens and borrowing them, however is a natural inclination. As such market caps should be put on those actions. At a minimum, implementing a cap on supplying the market token is needed, since no borrowing can be done if there are no underlying market tokens available.

Regardless of putting a cap on borrowing or lending tokens, the cap on depositing collateral should be removed.

For the case where a collateral is to be removed due to economic reasons, use the maximum loan-to-value (LTV) ration. If it is 0, then no extra collateral is to be added. Another solution is to convert the cap into a flag for enabled/disabled per collateral token.

Resolution: Resolved. Fix was implemented in **PR#118** and **PR#150**.

ABA: The cap on collateral deposit was removed and a cap on lending tokens was added.

[M-04] Collateral support cannot be completely dropped

> **Severity:** *Medium risk* *(Resolved)*
>
> **Context:** *state.clar*

Description

Granite governance can alter the supported collateral tokens by calling the `update-collateral-settings` function from the `state` contract. This function, however, does not provide a way to completely remove a collateral token.

In regard to removing or dropping support for a collateral token, there are two general situations:

1. a collateral has become unsuited from an economical point of view and removal is desired
2. a collateral has been discovered to have exploitable bugs or issues and must be immediately removed and not interacted with

In the second situations, immediate halting of any interaction with the collateral is required. But since there is no means to remove a collateral from the `state::collaterals` mapping, even with a minimum LTV ratio set, the token is still considered supported by liquidations, and adding/removing it as collateral.

As an observation, blocking further adding collateral can be done by setting the collateral cap to 0 but liquidating a position and users removing the collateral is not blocked.

All borrowing and lending system must have a mechanism to completely remove a formerly supported token, as to not risk any interaction with a potential faulty contract.

Recommendation

In the `state` contract, create a governance-callable-only function that removes a supported collateral token from the `collaterals` map, by deleting it.

Also add an appropriate action in the `governance` contract.

Resolution: Resolved, the recommended fix was implemented in **PR#**.

[M-05] Reducing collateral liquidation LTV ratio may instantly liquidate users

> **Severity:** *Medium risk (Acknowledged)*
>
> **Context:** ***state.clar****:346-370*

Description

Granite governance can alter the supported collateral tokens and their Loan-to-Value (LTV) ratios by calling the `update-collateral-settings` function from the `state` contract.

The LTV ration that is used to risk adjust the value of collateral when liquidating is referred to as liquidation LTV. The maximum LTV is the risk adjusted value of collateral when borrowing is initiated. These two exist as to make self liquidations in conjunction with oracle updates unprofitable for attackers.

If a token is deem unsuitable to be used as collateral from a market/economical perspective, or if the current LTVs are considered too high and are to be reduced, governance can call the `update-collateral-settings` with an appropriate maximum and liquidation LTV.

However, when doing this reduction, the collateral value of all existing borrowing positions that have guaranteed with it is instantly reduced, possibly resulting in users being instantly liquidated.

Recommendation

Modify the `update-collateral-settings` function to include a LTV ramp duration when changing the liquidation LTV. The ramp would be a linear decrease from when the update was done up to the desired value over the ramp period. If no ramp duration is set, then it would be instant. The ramp should only affect liquidations.

By doing it as so, users would still reach the liquidation point but not instantly, giving them a fair chance to unwind.

Resolution: Acknowledged by the team.

ABA: team acknowledged the issue as currently Clarity does not support block timestamp retrieval. This feature is planned to be added in the Nakamoto release.

Trust Machines: Implementation plan

Phase 1: off-chain ramping until block.timestamp is available

Phase 2: implement it on-chain, each collateral has got a ramp period (add it to the collaterals mapping).

[M-06] Lack of slippage on liquidations

Severity: *Medium risk* *(Resolved)*

Context: ***liquidator.clar****:28-107*

Description

When a liquidation is initiated, the liquidator calls the `liquidator::liquidate-collateral` function with the desired collateral token to liquidated and how much of the maximum debt repay amount he is willing to give for the discounted collateral. This function can be called either directly or through the `bundler` contract, to first update the oracle prices of the underlying collateral.

The amount of collateral (at a discount) a liquidator would receive and how much debt he must pay is determined by the `liquidator::liquidate` function. The two amounts depend, among others, on the collateral price and how much in position-health violation is the user.

Because of how the liquidation system is implemented, a liquidator may call `liquidator::liquidate-collateral` expecting a specific profit amount after liquidation, but can receive less, due to several factors:

- the user, intentionally or unintentionally, has front-run the liquidation with an adding more collateral to reduce the amount he is liquidated for
- the liquidator did not use the the `bundler::liquidate-collateral` function and is using a stale price. Subsequently, his liquidate transaction is executed in the block, after a new pyth oracle updated

In both of these cases, the liquidator receives less value than he was expecting. On certain cases, the liquidator may have not initiated the liquidation if he had up-to-date state information, due to lack of profitability.

Recommendation

Allow liquidators to pass a `minimum-collateral-received` amount when calling `liquidate-collateral`. If the resulting `collateral-to-give` from the `liquidate` call is not at least equal to this amount, then revert the liquidation. If liquidators do not wish to use this option, they can simply set it to 0.

Resolution: Resolved, the recommended fix was implemented in **PR#134**.

[M-07] Rounding in liquidations is against the protocol

Severity: *Medium risk* *(Resolved)*

Context: ***liquidator.clar****:77*

Description

Stacks smart contracts do not support floating point numbers arithmetic. As such, when implementing division operations within any protocol, of crucial importance is how rounding is considered.

To maintain protocol solvency over a long period of time, for borrowing and lending contracts, all rounding must be against the user and in favor of the protocol.

Granite correctly respects this idea, with one exception, in liquidations.

When a position is liquidated via a `liquidator::liquidate-collateral` call, the debt shares that will be deducted out of the users' total shares will be incorrectly rounded up:

```
(paid-shares (contract-call? .math convert-to-debt-shares repay-amount true))
```

Subsequently the potentially inflated share value is then deduced from total user debt shares and total protocol debt shares in `state::update-liquidate-collateral-state`:

```
(map-set positions user {debt-shares: (- (get debt-shares position) paid-shares),
    collaterals: (get collaterals position)})
(var-set total-debt-shares (- (var-get total-debt-shares) paid-shares))
```

In this case, because of the incorrect rounding, several bad situations can appear:

- with each liquidation, the protocol and lenders lose more and more revenue. While it is not much per one liquidation, over time, it will add up.
- highly unlikely, but a case where a user cannot be liquidated since the subtraction of this debt shares `(- (get debt-shares position) paid-shares)` would overflow due to the paid-shares being off-by-one. In this case, a liquidator may do smaller liquidations to avoid triggering the issue
- for markets with low usability, the last borrower with an opened position cannot be liquidated since the total debt reduction `(- (var-get total-debt-shares) paid-shares)` would underflow

Recommendation

Round down the repaid shares on liquidations. by changing the `round-up` flag when calling `math::convert-to-debt-shares` to `false`.

Resolution: Resolved, the recommended fix was implemented in **PR#127**.

[M-08] Governance proposals never expire and can't be rejected

Severity: *Medium risk* *(Resolved)*

Context: ***governance.clar***:*596-597,179-194* ***meta-governance.clar***:*229-230,74-89*

Description

In this situation, two issues regarding proposals are of concern when combined:

The first issue is that, when a proposal is created either related to Granite configuration through the `governance` contract, or related to changing the multisig ownership through the `meta-governance` contract, there is no way to add an expiration date, or deadline, to a proposal.

Proposal only contain the action to be done, the number of approvers and deniers, and if it was completed.

```
;; Governance proposal
(define-map governance-proposal uint (tuple
  (action uint)
  (approve-count uint)
  (deny-count uint)
  (completed bool)
))
```

As such, any outdated proposal from the past, can still be executed in the future, even with years apart.

The second issue that increases the severity is that, although both the `governance` and `meta-governance` contracts do support proposal denials, none fully or correctly implement them.

In the `governance` contract, the `deny` function exists, but after users do deny, it incorrectly tries to approve the proposal by calling the `execute-if-approve-threshold-met` instead of the `deny-proposal-if-deny-threshold-met` function.

```
;; deny proposal if the deny threshold is met
(try! (execute-if-approve-threshold-met proposal-id))
```

In the `meta-governance` there is the same issue with the call to `execute-if-approve-threshold-met` in the `deny` function, but in this contract the entire deny functionality is missing.

Although denying a function does not invalidate it, it does block the vote, as it can't be changed afterwards. This in itself would be a minor inconvenience but coupled with the fact that a proposal never expires makes it an issue.

It allows any future governance holder, compromised or simply in disagreement with others, to take any past proposal and accept it, possibly making it valid and executed.

Recommendation

Add a deadline to proposals. The deadline should be in time units when Nakamoto is launched, but currently, for validation and testing it can use block height.

Also, correctly implement the deny functionality in both governance related contracts. If no deny is actually desired, but only to block the vote, remove the `execute-if-approve-threshold-met` call in both deny function calls.

Resolution: Resolved, the recommended fix was implemented in **PR#133** and **PR#files**.

[M-09] Governance multisig can be corrupted by adding existing or removing inexistent members

Severity: *Medium risk* *(Resolved)* *[PoC]*

Context: ***meta-governance.clar***:*102-120,154*

Description

Through the `meta-governance` contract, principals can be added to or removed from the governance multisig member list.

Adding and removing, however, do not validate that they are adding an already existing principal or removing an already removed (or inexistent) principal. In both cases the proposals are created and can pass. The number of multisig members would be increased or decreased but without there actually being the indicated number of principals allowed.

The `execute-update-governance-multisig` function is responsible for updating the governance member list

```
(define-private (execute-update-governance-multisig (proposal-id uint) (action
    uint))
  (let ((governance (unwrap-panic (map-get?
      update-governance-multisig-proposal-data proposal-id))))
    (if (is-eq action ACTION_ADD_GOVERNANCE_MULTISIG)
     (begin
      (map-set governance-accounts governance true)
      (var-set governance-accounts-count (+ (var-get governance-accounts-count) u1))
      SUCCESS
     )
      (if (is-eq action ACTION_REMOVE_GOVERNANCE_MULTISIG)
        (begin
          (map-delete governance-accounts governance)
          (var-set governance-accounts-count (- (var-get governance-accounts-count)
              u1))
          SUCCESS
        )
        ERR-INVALID-ACTION
      )
    )
  )
)
```

We can see that when adding members, the `(map-set governance-accounts governance true)` response is not checked for `true`, that it actually did add a member. Also, when removing a principal, the deletion `(map-delete governance-accounts governance)` operation is not checked for a `true` response, which would indicate a successful removal.

The same lack of duplicated principals being present also exists in the `set-governance-multisig` function, which sets the initial list of principals on initialization.

This corruption can result in a multisig not being ever able to reach the approval threshold, permanently blocking all operations. Some examples bad scenarios:

- a one multisig principal proposal mistakenly re-adding himself would block any other operations
- a two multisig principal proposal incorrectly removing an inexistent principal would subsequently allow the creation an acceptance of proposals with only 1 approve. A 50% approval rate instead of a 66%
- the same proposal was mistakenly created twice and over time it was accepted twice with the false impression that the second operation would have no side effect

Recommendation

In the `meta-governance::execute-update-governance-multisig` function, check the return of the `map-set` and `map-delete` operations to always be true, otherwise revert the execution as the proposal was either created with invalid principals or was a duplicated proposal.

It is not enough to validate when creating the proposal if a member exists or not, as two or more identical proposal can be created at the same time where both would add or remove the same principal. Both can be accepted and would lead to the same issue appearing.

Resolution: Resolved, the recommended fix was implemented in **PR#**, **PR#126** and **PR#files**.

[M-10] account-health does not take into consideration pending interest

> **Severity:** *Medium risk* *(Acknowledged)* *[PoC]*
>
> **Context:** ***liquidator.clar****:156-178*

Description

Liquidators determine if a user is liquidatable by his health factor. On-chain, the function `liquidator::account-health` returns this value.

However, `account-health` does not take into consideration any pending interest owed by borrowers since the last time interest was accrued.

Accruing interest is called on most operations in Granite but situations where it lags can appear due to low market activity. In those situations, any off-chain liquidator using the current `account-health` function will see an invalid. more healthy status.

This may lead to positions not being liquidated since they are not perceived as unhealthy without the added interest.

Recommendation

Create a separate `account-health` function with the sole purpose of being used by off-chain liquidators to determine the up-to-date health factor of a user.

Within it, simulate the `math::convert-to-debt-assets` logic as if the pending interest has been added to it; as if `liquidator::accrue-interest` was previously called.

Since this function would live in the `liquidator` contract and only be called off-chain, there would be no execution fees associated with it.

Common code between the two versions of `account-health` should be moved to a private function.

Resolution: Acknowledged by the team.

Trust Machines: for off-chain calculations, our Math SDK already takes this into account.

[M-11] Anyone can liquidate a position with bad debt

Severity: *Medium risk* *(Resolved)* *[PoC]*

Context: ***liquidator.clar***

Description

In Granite, any liquidator can create bad debt, contrary to what the documentation indicates. Meaning that market participants can call `liquidator::liquidate-collateral` on a liquid position, leaving the liquidated with no collateral but debt.

Recommendation

Implement the mentioned feature.

Resolution: Resolved. Fix was implemented in **PR#119**.

ABA: team decided to not implement this feature, although a fix was initially made and validated in PR#119.

3.4 Low Severity Findings

[L-01] Missing getter for deposit and withdrawal status

Severity: *Low risk* *(Resolved)*

Context: ***state.clar***

Description

The `state` contract contains several flags that enable or disable system operations, such as liquidations or depositing collateral. Most of these flags are accompanied by getters in order for other components of the protocol to use or third parties that would integrate with the system.

However, two features, market token withdrawals — the `withdraw-asset-enabled` flag — and deposits — the `deposit-asset-enabled` flag — do not have getters.

As data variables in Clarity cannot be access without specific getters, any third party system that would integrate with Granite cannot determine if providing or withdrawing market tokens is enabled, before interacting with the system.

Recommendation

Create a `is-withdraw-asset-enabled` and `is-deposit-asset-enabled` function in the `state` contract.

Resolution: Resolved, the recommended fix was implemented in **PR#132**.

[L-02] Governance multisig member count should not be allowed to reach 0

Severity: *Low risk* *(Resolved)*

Context: ***meta-governance.clar***:*110-115,235-243*

Description

With the current implementation of the `meta-governance` multisig contract, the governance can completely remove all its members, effectively losing the ability to set any Granite configuration or pause the system if needed.

Recommendation

In the `meta-governance::execute-update-governance-multisig` function, when execution the remove action, check if `governance-accounts-count` is currently 1 and do not allow the removal to be executed if so.

In the `meta-governance::initialize-governance` function, after the `(map set-governance-multisig governance-multisigs)` call, retrieve and verify that the `governance-accounts-count` count is at least 1 or revert.

Resolution: Resolved, the recommended fix was implemented in **PR#125**.

[L-03] Interest accrual cannot be suspended

> **Severity:** *Low risk* *(Resolved)*
>
> **Context:** *linear-kinked-ir.clar*

Description

Project documentation indicates that governance can suspend interest accrual. This feature is not implemented within the codebase.

In unlikely situation where issues are found with the accrual contract, it cannot be blocked while the issue is mitigated.

Recommendation

Change the documentation to indicated that it will be added in a further version or remove the mention.

Or add the feature. This would require changes in the `state` (getters/setters), a specific action in the `governance` contract and modifying `linear-kinked-ir::accrue-interest` to check if it is paused, to simply pass-through the inputs as outputs, as to not block the system.

Resolution: Resolved, the recommended fix was implemented in **PR#162**.

[L-04] Protocol upgradeability cannot be freezed

> **Severity:** *Low risk* *(Resolved)*
>
> **Context:** *governance.clar*

Description

With regards to upgradeability, project documentation indicates that:

> At any point in time, by toggling a specific irreversible flag, the governance can block any upgrades to the protocol and lock the contracts to a specific immutable implementation.

However, this feature is currently not present in the codebase. Although upgrades can be blocked by governance by initiating a proposal via the `governance::initiate-proposal-to-set-market-feature` function with the `ACTION_SET_UPGRADES` action and flag set to `false`, governance can create a new proposal afterwards to unblock upgrades via the same parameters but with the flag set to `true`.

Protocol functionality regarding centralization should be clearly and correctly indicated in the project documentation.

Recommendation

Either modify the documentation to reflect current implementation or modify the `governance` contract to only allow disabling upgradeability.

Resolution: Resolved, the recommended fix was implemented in **PR#143**.

[L-05] Collateral tokens with a maximum LTV of 0 should not be accepted as collateral

Severity: *Low risk* *(Resolved)*

Context: ***borrower.clar****:110-151*

Description

When the protocol decides that a specific token is unsuited to continue being used as collateral, from an economical point of view, they can initiate its removal by setting a borrow (maximum) loan-to-value ration (LTV) of 0.

This implies that when borrowing, the specific collateral token value would be considered 0. A different collateral LTV is used when considering liquidations.

Setting the maximum LTV to 0 while steadily decreasing the liquidation LTV is commonly used as to not instantly liquidated users. Thus, a maximum LTV of 0 can exist at the same time as a non-zero liquidation LTV ratio.

However, a user would still be able to add collateral to the pending-removal collateral token via the `borrower::add-collateral` function, increasing the health of his already existing positions.

Recommendation

In the `add-collateral` function, do not allow collateral with a maximum LTV ration of 0 to be added.

Resolution: Resolved, the recommended fix was implemented in **PR#124**.

[L-06] Protocol reserve percentage can be set to over 100%

Severity: *Low risk* *(Resolved)*

Context: ***state.clar****:519-527*

Description

The protocol reserve percentage defines what percentage of the gained interest is set aside for the protocol. This value is set by governance while calling `state::set-protocol-reserve-percentage`.

However, there is no sanity check on the new value of the reserve percentage. A percentage over the equivalent of 100% would block several operations within Granite due to overflows. Operation such as liquidating or borrowing are blocked due to an underflow in the `linear-kinked-ir::accrue-interest` function.

Recommendation

Validate in the `governance::initiate-proposal-to-update-protocol-reserve-percentage` function that the new value of the protocol reserve percentage is less then the equivalent of 100%.

Normally such checks are done exactly before committing them, meaning in the `state` contract but as the `state` contract is highly imported throughout the codebase, they can be done in the `governance` contract.

Resolution: Resolved, the recommended fix was implemented in **PR#123**.

[L-07] Small loans may be unprofitable to liquidate

> **Severity:** *Low risk* *(Resolved)*
>
> **Context:** *liquidator.clar borrower.clar*

Description

A general issue for borrowing and lending protocols is regarding the profitability of liquidating small loans.

If the borrowed amount and the collateral deposited are too low, then it is possible for the discounted collateral that a liquidator receives from a liquidation might not be enough to cover the fee execution cost of the liquidation operation itself.

In this scenario, with no incentive to liquidate loans that are getting underwater, then the protocol will continue to accumulate interest and subsequently bad debt.

The problem of small positions has been discussed extensively in public. Protocols have generally chose one of two approaches:

1. Add a minimum borrow amount as to ensure users have a sufficient large amount of backing collateral.
2. Leave the system as it is. Small liquid positions will accumulate over time but in practice, these small positions have not been extensively proven to affect the markets. If needed, governance itself would liquidate them at a loss. For example, both Euler v1 and Euler v2 follow this approach.

Recommendation

As mentioned in the description, one solution is to implement a minimum borrow amount. This would however, increase code complexity and can be added later, if actually needed.

Thus, the recommendation is to acknowledge the possibility of this issue and, if actually needed, have governance liquidated small positions by itself. This comes with the tradeoff of paying fees to liquidated positions.

Resolution: Resolved, the recommended course of action was taken.

Trust Machines: Governance can liquidate small loans, there is little evidence that such an issue can lead to major service disruption or DOS.

[L-08] Liquidations are blocked if collateral price reaches 0

Severity: *Low risk (Resolved)*

Context: *liquidator.clar:194*

Description

When a user is liquidated, if the oracle returns 0 as the liquidated collateral price, then liquidation will revert when calculating the amount of collateral to give to user.

The revert is in the `liquidator::calc-collateral-to-give` function when dividing by `collateral-price`:

```
(collateral-amount (try! (safe-div (* repay-amount-with-discount resolution-factor)
    collateral-price)))
```

A faulty oracle, or an extreme market event may lead to this situation manifesting. In this scenario, liquidators cannot liquidate users have that specific collateral as the payment.

Of particular relevance are users that have only this collateral installed, as they can never be liquidated.

Users that have 2 or more collaterals can be partially liquidated by having liquidators choose a different reward collateral, but will ultimately leave the users with only the last collateral, and most likely debt, without the possibility to close his position (even at a loss to the protocol)

Recommendation

In Granite, the oracle that will be used is `pyth` when the latest version will be on Stacks mainnet, after Nakamoto.

At that point, thoroughly validate that the only case in which the oracle would return 0 is if the token price is actually 0. This is to distinguish between a faulty implementation or actual 0 price.

Modify the liquidation logic so that if the collateral price is 0, then the entire collateral balance of the liquid account to be seized. A side effect of this alteration is that collateral with no value can be claimed through liquidation. The same effect also accrues if the liquidation LTV is set to 1 (it cannot be set to 0).

It is somewhat debatable if the liquidator's ability to claim collateral even when it holds no value align with the logic of the liquidation process. However, removing dust collateral and closing the outstanding positions could only be done as so if needed.

Resolution: Resolved, the recommended fix was implemented in **PR#152**.

[L-09] All liquidity related calls should go through the bundler contract to have up-to-date prices

Severity: *Low risk (Resolved)*

Context: *borrower.clar liquidator.clar*

Description

The current system design allows operations allows such as borrowing, removing collateral and liquidating to be called directly without a price update.

A direct call is done either through the `borrower` contract or the `liquidator` contract.

As a utility contract, `bundler` allows the exact same three operations to be called but with an update to the pyth oracle prices.

Any user call that does not pass through `bundler` on the three mentioned operations risk using stale prices, resulting in a different expected outcome.

Example Scenario:

- `alice` sees that the oracle was updated in the last block (approx. 6 seconds ago) so she calls `borrower::borrow` directly to save on fees
- in the same block but before `alice`, `eve` has called `bundler::liquidated-collateral` with a higher (but still valid) market token pyth update price, in order to increase her liquidation profit of `joe`
- `alice` now has a slightly different health status that what she was expecting due to the price variation

Another example would be in reverse

- `alice` sees that the oracle was updated in the last block (approx. 6 seconds ago) so she calls `liquidator::liquidated-collateral` to liquidate `joe` directly to save on fees
- in the same block but before `alice`, `eve` has called `bundler::borrow` with a lower (but still valid) market token pyth update price, in order to increase the amount of tokens she can borrow
- `alice` now has liquidated `joe` but received a slightly less amount as profit

Recommendation

While the team may decide to leave the possibility to execute liquidations, borrowing and removing collateral directly, they should explicitly advise users to always use the `bundler` versions with price update to avoid any unwanted scenario.

Consider adding constraints so that the mentioned operations are not allowed to be executed outside the bundler.

Resolution: Resolved. Fix was implemented in **PR#131**.

ABA: the team decided to completely remove the bundler and modify all actions that require up-to-date price to also accept an optional pyth VAA price update buffer.

[L-10] Stacks reorgs may affect governance proposals

Severity: ***Low risk*** *(Resolved)*

Context: ***governance.clar****:180,184* ***meta-governance.clar****:75,79*

Description

After the Nakamoto update, the Stacks network will also produce blocks in-between Bitcoin blocks (5-6 seconds per block). Up until the moment the Stacks state is committed on the Bitcoin chain, although unlikely, block reorgs can occur in theory.

When created, governance proposals are given an incremental ID starting from `0`. Members then vote on a proposal by identifying it by its ID. Consider the following scenario:

- two proposal are created in consecutive blocks
- proposal 0: `setting sBTC borrow LTV to 95%`
- proposal 1: `setting sBTC borrow LTV to 85%`
- group A votes approve to proposal 0
- group B votes approve to proposal 1
- a block reorg happens of a few blocks, which shuffles the creation of the 2 proposals in reverse
- as the ID is generated incrementally, **proposal 0** becomes `setting sBTC borrow LTV to 85%` and **proposal 1** becomes `setting sBTC borrow LTV to 95%`
- as the blockchain reorg left the votes as they were, each group has their votes now reversed, possibly leading to an unwanted situation

The same issue also applies when adding or removing governance members within the `meta-governance` contract.

Recommendation

In the `create-proposal` function of both the `governance` and `meta-governance` contracts, generate the proposal ID using a method that is dependent on its contents. For example, creating a hash on the publication content (just the action is not enough).

Resolution: Resolved, the recommended fix was implemented in **PR#133**.

[L-11] Add extra protection against self-liquidations

> **Severity:** ***Low risk*** *(Resolved)*
>
> **Context:** ***liquidator.clar borrower.clar***

Description

A generally known vulnerability with borrowing and lending protocols is related to self-liquidations during a market price updated.

An attack abusing this would unfold as:

- an attacker would look for the oracle price update transaction, one which would reduce the price of the collateral
- sandwich the oracle price update transaction:
  - front-run it with a deposit collateral plus borrow
  - back-run it with a self liquidation

  - remove remaining collateral

Note, the attack only works if debt socialization is enabled.

The attack is only profitable if the entire collateral is retrieved plus a difference. Flash loans are usually used to help increase profits. Also, for systems relying on oracles like pyth, as the case of Granite, the attack has a higher chance of being executed as anyone can update the price with a valid range of potential prices within a given time-frame (1 minute by default on the pyth network).

This type of attack is also known as oracle frontrunning.

To mitigate the attack, Granite already implemented differential loan-to-value ratios (LTV). When a user borrows, his collateral is evaluated at a borrow LTV and when liquidated, the same collateral is evaluated at a higher LTV. This is done to ensure that users cannot be instantly liquidated due to the valuation gap between the borrow valuation and the liquidation valuation.

Besides the already mentioned mitigation, Granite can also implement other mechanism as to further reduce the efficiency of the attack.

Recommendation

Oracle frontrunning attacks can't be fully mitigated. However, to reduce the risk of such an attack an extra mechanism can be implemented of not allowing a user to be liquidated in the same block as when he last borrowed.

This mechanism would specifically block flash-loan based attacks, since an attacker would be unable to borrow, liquidated and pay back the loan within the same transaction even with an unlikely huge movement in price.

Since borrowing cannot be done if the collateral value adjusted by the borrow (maximum) LTV, is less than the debt valuation, the chance in practice of a normal happy-path user avoiding liquidation (for 1 block) in this case is extremely unlikely and can be considered acceptable.

Resolution: Resolved, the recommended fix was implemented in **PR#153**.

[L-12] There are no sanity checks when updating interest accrual settings

> **Severity:** ***Low risk*** *(Resolved)*
>
> **Context:** ***governance.clar****:538-553*

Description

The linear kinked interest model, represented by the `linear-kinked-ir` contract, configuration parameters can be changed by calling `linear-kinked-ir::update-ir-params`. After the initial deployment and initialization, governance can change the interest settings by calling the `governance::initiate-proposal-to-update-interest-params` with the corresponding new values.

There are no validations done on any of the new parameters, in either situation. If invalid values would be mistakenly passed, the interest calculations will be off, leading to incorrect interest accumulation.

Recommendation

In the `initiate-proposal-to-update-interest-params` function, verify that `utilization-kink-val` verify does not surpass 100% utilization equivalent, which would mean the kink is never reached.

Also, verity the value when deploying the contract.

Resolution: Resolved, the recommended fix was implemented in **PR#122**.

[L-13] Avoid using tx-sender for sensitive operations

> **Severity:** ***Low risk*** *(Resolved)*
>
> **Context:** Global

Description

Throughout the contract there are instances where `tx-sender` is used instead of `contract-caller` or passing the user address.

Doing this, users that fall to phishing scams and interact with malicious contracts can unwittingly interact with Granite and execute operations.

For example, if a user interacts with a malicious contract, that contract can then call `borrower::borrow` and make the user randomly do a borrow (the user will get the borrowed amount) but a borrow was made without the user agreeing.

The malicious contract could also call `borrower::remove-collateral` to make the user remove collateral, making him temporarily less healthy, until he re-adds the collateral.

This cannot happen without user interaction.

Recommendation

Extensive code alterations are needed so that every single instance of the `tx-sender` must be changed accordingly and depending on the context of the operation.

More specifically:

1. in the `borrower` contract modify all the functions where `tx-sender` is used: `borrow`, `repay`, `add-collateral`, `remove-collateral`, `get-user-collaterals-value`, `iterate-collateral-value` and `remove-user-collateral` to also accept a user principal address that is to be used instead of `tx-sender`
2. permissionless call to the public function: `borrow`, `repay`, `add-collateral`, `remove-collateral` must not be allowed at this point as anyone would be able to borrow/repay/add-collateral in anyone else's name. An authorization mechanism is required that would allow only the bundler contract to call
3. a `add-collateral` version must be added to the `bundler` contract, as again, no public function must be permissionless callable in the `borrower` contract

4. in the `liquidator` contract, the same as borrow, modify the `liquidate-collateral` function to accept an argument representing the liquidator, make the function callable only by an authorized contract (the `bundler`) and change all `tx-sender` instances with the passed liquidator variable
5. in the `bundler` contract, pass the `contract-caller` to the `borrower` and `liquidator` public functions. Example:

```
(define-public (borrow (pyth-price-feed-data (buff 8192))
   (amount uint))
  (begin
    (try! (update-pyth pyth-price-feed-data))
-   (contract-call? .borrower borrow amount)
+   (contract-call? .borrower borrow amount contract-caller)
  )
)
```

1. if intended (although not recommended and detailed in another issue) also create public versions that do not require to update the pyth oracle in the `bundler` contract
2. in the `liquidity-provider` contract change `tx-sender` to `contract-caller` in all occurrences
3. in the `linear-kinked-ir` contract, use `contract-caller` instead of `tx-sender` in the `update-ir-params` function. Likewise in the `pyth-oracle` contract
4. in the `governance` and `meta-governance` contract directly modify all instances of `tx-sender` to `contract-caller` with the sole exception of the `contract-deployer` variable/constant
5. changes to the `governance::execute-state-reserve-action`, that also cover this issue, were mentioned in a different issue
6. in the `state` contract, in the `deposit-to-reserve` function, change the first `tx-sender` in the transfer call to the `contract-caller`
7. in the `state` contract again, in the `transfer` function, remove the `(is-eq tx-sender sender)` condition in the `asserts!`. No other changes in the `state` contract are needed with regards to `tx-sender`

Resolution: Resolved, the recommended fix was implemented in **PR#141**.

[L-14] Collateral discount and liquidation LTV incompatible values can crash liquidations

Severity: ***Low risk*** *(Resolved)*

Context: ***liquidator.clar****:127-131* ***state.clar****:346-370*

Description

When a liquidation is done, the maximum repayable amount is calculated so that the position health reaches the value 1 where risk adjusted collateral is equal to debt.

The `liquidator::liquidate` calculates the maximum amount, by first calculating a denominator as:

```
(denominator (-
    resolution-factor
    (try! (safe-div (* (+ resolution-factor liquidation-discount)
        collateral-liquid-ltv) resolution-factor))
  )
)
```

Mathematically, the code models the formula

$\text{denominator} = 1 - (1 + \text{liqDiscount}) \times \text{liqLTV}$

This formula, however, introduces an extra limitation as it cannot be a negative number. Formalizing this, the following condition needs to be uphold:

$1 - (1 + \text{liqDiscount}) \times \text{liqLTV} > 0$

$(1 + \text{liqDiscount}) \times \text{liqLTV} < 1$

$\text{liqLTV} < \frac{1}{1+\text{liqDiscount}}$

For example, if the liquidation LTV is 90% and collateral discount is 15% then:

$0.9 < \frac{1}{1+0.15}$

$0.9 < \frac{1}{1.15}$

$0.9 < 0.869 \implies \text{false}$

The result is `false` and liquidation will revert with an underflow runtime error.

Recommendation

As the liquidation mechanism was designed to function with the premise of arriving at maximum to 1 health value, the condition $\text{liqLTV} < \frac{1}{1+\text{liqDiscount}}$ must be enforced when setting the liquidation LTV and discount for each collateral.

Ideally this check should be done in the `state::update-collateral-settings` function, but it can be implemented, to save execution costs, in the `governance::initiate-proposal-to-update-collateral-settings` function.

Resolution: Resolved, the recommended fix was implemented in **PR#145**.

[L-15] Liquidations do not prioritize the lowest LTV asset

Severity: ***Low risk*** *(Resolved) [PoC]*

Context: ***liquidator.clar****:28-107*

Description

When a liquidation is done, the liquidator specifies which collateral token to receive at a discount. Because each collateral token has a different Loan-To-Value (LTV) ratio, a liquidation extracting collateral with a high LTV, while leaving the lower LTV collateral, can result in a position becoming unhealthier post-liquidation.

The likelihood of such a scenario is however high, since, economically, less volatile collateral will be preferred by liquidators and at the same time will have the highest liquidation LTV set due to the same reasoning.

Such a position would require multiple liquidations to increase its health score. This slightly increases the overall debt the user has by a factor of the interest per block until fully liquidated. Also, in a situation where the lowest LTV collateral is rapidly decreasing in price, since liquidators will priorities any other higher LTV collateral, there is an additional risk of the price of such a collateral to further decrease while not being seized from the protocol, risking the accumulation of bad debt.

Recommendation

Fully mitigating the issue would require redesigning liquidations to not give the option of what collateral to seize and directly choose the lowest LTV collateral.

This solution would need to be accompanied by a logic to allow multiple token liquidations simultaneously since otherwise liquidators would risk not being profitable in situations where a borrower would abuse the system to delay liquidation. For example: user has the maximum number of available collaterals `M`. The first `M-1` that have the lowest LTV are only dust amounts while the highest LTV is the only practical one the borrower. A liquidator would need to execute `M` liquidations in order to be in profit.

Another at-hand, but not ideal, idea would be to allow only liquidations that would increase the position health or revert otherwise. Reverting liquidations in case the after position health is less then initial is not a viable option due to a possible, but highly unlikely, Denial-of-service (DOS) scenario where market conditions would bring the prices/values of collateral down at such an extend that the position will never be healthier after a liquidation, thus will never be liquidatable and continuously accumulating bad debt.

A straightforward solution is not available due to advantages and disadvantages of each approach.

The recommended suggestion is to mitigate the issue by carefully selecting collateral tokens that do not have a high difference between liquidation LTV, accompanied by monitoring and reactive strategies.

This already would be done for when monitoring for black-swan type events. Example, in case of high market volatility, monitor position liquidations and, in the event of drastic price deduction for a specific collateral, the protocol can apply one of several options: not allow liquidations, change LTV, immediately remove a collateral or simply liquidating the positions themselves at a loss with debt socialization (if implemented). Each situation, of course, would be thoroughly analyzed.

Resolution: Resolved, the recommended course of action was taken.

ABA: the team is now aware of the behavior and will monitor the system to see if issues do arise, as recommended.

3.5 Informational Findings

[I-01] Reuse SUCCESS constant

> **Severity:** *Informational* *(Resolved)*
>
> **Context:** ***state.clar***:*449*

Description

In the public `transfer` function from the `state` contract `(ok true)` is returned instead of the already defined constant SUCCESS. Besides increasing readability, reusing the SUCCESS constant will also help reduce code size by 2 bytes.

There are other locations in the codebase where the SUCCESS may be reused instead of `(ok true)` but in those particular cases the response is part of a boolean function. It is semantically different from success, as such, it should not be changed there also.

Recommendation

Replace `(ok true)` with SUCCESS in the indicated location.

Resolution: Resolved, the recommended fix was implemented in **PR#135**.

[I-02] Remove unused constants

> **Severity:** *Informational* *(Resolved)*
>
> **Context:** ***bundler.clar***:*7-12* ***linear-kinked-ir.clar***:*11* ***pyth-oracle.clar***:*13-14* ***borrower.clar***:*18* ***governance.clar***:*79*

Description

Throughout the codebase there are declared constants that are not used.

- `bundler`: ERR-INVALID-BUFFER and SUCCESS
- `linear-kinked-ir`: ERR-DIVIDE-BY-ZERO
- `pyth-oracle`: ERR-ASSET-ALREADY-RECORDED and ERR-PYTH-READ-FAILURE
- `borrower`: ERR-COLLATERAL-NOT-SUPPORTED
- `governance`: ERR-NOT-GOVERNANCE and ERR-STATE-CONTRACT-MISMATCH

Recommendation

Remove them to reduce smart contract size. Also, since the constants are errors, change, if needed, the error value of adjacent constants to not have gaps.

Resolution: Resolved, the recommended fix was implemented in **PR#135**.

[I-03] borrowing::borrow can be simplified

> **Severity:** *Informational* *(Resolved)*
>
> **Context:** ***borrower.clar***:*38,41,50*

Description

In the `borrow` function of the `borrower` contract, the `user-posted-collaterals` and `position-collaterals` variables are identical and equal to `(get collaterals position)`.

The duplication increases code size, gas costs, and adds an extra operation for any borrow call.

Recommendation

Remove the `user-posted-collaterals` variable and reuse the already existing `position-collaterals` in its place.

Resolution: Resolved, the recommended fix was implemented in **PR#149**.

[I-04] state::update-borrow-state can be simplified

> **Severity:** *Informational* *(Resolved)*
>
> **Context:** ***state.clar***:*691-710*

Description

The `state` contract `update-borrow-state` function can be simplified. It retrieves the `amount` element from the `borrow-state` three times and the `user` element two times.

Recommendation

Change the `begin` to a `let`, declare the `amount` and `user` variables and reuse them in the code. Example declaration snippet:

```
(let (
    (amount (get amount borrow-state))
    (user (get user borrow-state))
  )
```

Resolution: Resolved, the recommended fix was implemented in **PR#149**.

[I-05] state::update-liquidate-collateral-state can be simplified

> **Severity:** *Informational* *(Resolved)*
>
> **Context:** ***state.clar***:*771-813*

Description

In the `state::update-liquidate-collateral-state` function, there are a few redundancies that can be removed.

The `user` variable is declared and set on line 787 but above it, the same tuple element is retrieved twice and bellow it in the `user-collaterals` `map-set`, it is again retrieved.

Also, the `collateral-to-give` variable is declared and set on line 789 but on line 803 it is again retrieved instead of being reused.

Recommendation

Move the local `user` variable declaration first in the `let` declaration and reuse in all the previously mentioned positions.

Reuse the local `collateral-to-give` variable in the map-set operation.

After the above two changes, compress the map-set user-collaterals to a single line, as it now has a sufficient length to do so without losing readability.

All the mentioned changes reduce the code line count by 3 and code size by 159 bytes while maintaining readability.

Resolution: Resolved, the recommended fix was implemented in **PR#149**.

[I-06] governance::execute-state-reserve-action can be simplified

> **Severity:** *Informational* *(Resolved)*
>
> **Context:** ***governance.clar****:272-275*

Description

The `governance::execute-state-reserve-action` function is responsible for executing both withdrawals and deposits into the Granite market.

The deposit operation is done as so:

```
(if (is-eq action ACTION_DEPOSIT_TO_RESERVE)
  (begin
    (as-contract (try! (contract-call? .mock-usdc transfer amount (as-contract
        tx-sender) sender none)))
    (contract-call? .state deposit-to-reserve amount)
  )
```

It initially sends the to-be-deposited funds from the governance multisig to the sender of the transaction and then calls the `state::deposit-to-reserve`. The deposit function takes the funds from the caller.

This logic, although correct, can be simplified be rewriting it to directly call the `deposit-to-reserve` from with a `as-contract` context while achieving the same outcome.

Recommendation

Implement the mentioned changes. Example change:

```
    (if (is-eq action ACTION_DEPOSIT_TO_RESERVE)
-      (begin
-        (as-contract (try! (contract-call? .mock-usdc transfer
   amount (as-contract tx-sender) sender none)))
-        (contract-call? .state deposit-to-reserve amount)
-      )
+      (as-contract (contract-call? .state deposit-to-reserve
   amount))
```

Resolution: Resolved, the recommended fix was implemented in **PR#148**.

[I-07] Use errors instead of panicking with slight codebase simplifications

Severity: *Informational* *(Resolved)*

Context: ***borrower.clar***:*195,237* ***liquidator.clar***:*55,235* ***state.clar***:*162,399,415,559* ***math.clar***:*60* ***governance.clar***:*208,267,285,297*

Description

Throughout the codebase there are instances where `unwrap-panic` is used instead of `unwrap!` with a custom error.

Ending the execution in a panic manner results in a runtime error appearing. Runtime errors cannot be handled by the caller and do not encapsulate any meaningful information about the execution, as such, they are discouraged.

The following changes can increase code readability while also reducing code size:

1. in the `borrower` contract on the functions `get-user-collaterals-value` use the `ERR-NO-POSITION` error when users does not have a position
2. for the functions `liquidator::liquidate-collateral` set an error relating to failed to retrieve oracle price
3. in the `state` contract, include `unwrap-panic` in `free-liquidity` function body itself as every call to it prepends that
4. in the `state::available-liquidity` function, call the `free-liquidity` function instead of calling `get-balance` directly, reducing code size and removing the extra `unwrap-panic` call
5. in the `governance` contract there are proposal action groups that can be initiated with a specific range of actions but there are no validations done when they are called that the passed actions are in fact valid. The operations do revert on invalid actions because of retrieving the proposal specific data via `unwrap-panic` but in a generic manner.

Example when executing `initiate-proposal-to-set-market-feature` the action can be one of several, but if not correctly passed, the operation reverts in `execute-state-set-feature-flag` at the following operation:

```
(let ((flag (unwrap-panic (map-get? set-market-feature-proposal-data proposal-id)))
```

The other instances of this, where execution can reach and revert end up in the `execute-state-set-market-flag`, `execute-state-reserve-action`, `execute-state-set-allowed-contract` and `execute-update-guardian` functions.

In all the above-mentioned functions, add a meaningful error message.

The other `unwrap-panic` locations in the `governance` contract can be left as they are since there is no code path to them that would make them revert.

Recommendation

Implement the mentioned changes in each situation. There are several other instances of `unwrap-panic` left which if modified, would add more code complexity for too little benefits, as such are not part of the recommendations.

Resolution: Resolved. Fix was implemented in **PR#156**.

ABA: for remediation point 5, the team opted to add checks on the action value when creating the proposals, thus the panics will never be reachable.

[I-08] governance::execute-update-guardian can be improved

> **Severity:** *Informational* *(Resolved)*
>
> **Context:** ***governance.clar****:296-306*

Description

The `governance::execute-update-guardian` function handles executing either removal or addition of guardians.

The `if` branches where it adds or removes members can be rewritten to remove the `begin` altogether, since the `map-set` and `map-delete` functions themselves return booleans. This instead of the `begin` statement and returning SUCCESS the code cam be rewritten as:

```
     (if (is-eq action ACTION_ADD_GUARDIAN)
-     (begin (map-set guardians guardian true) SUCCESS)
+     (ok (map-set guardians guardian true))
       (if (is-eq action ACTION_REMOVE_GUARDIAN)
-        (begin (map-delete guardians guardian) SUCCESS)
+        (ok (map-delete guardians guardian))
         ERR-INVALID-ACTION
```

Another potential issue is that the return of `map-set` and `map-get` is never checked. This means that if the proposal removes an inexistent principal or re-adds an existing guardian, the return value is `false`.

Recommendation

As the code currently is, there are no issues with the mentioned scenario apart for the need of a new, correct, proposal. If the development team wishes to validate the return of `map-set` and `map-get`, then the `begin` should be kept and the validation done within it, otherwise the mentioned simplification can be implemented.

Resolution: Resolved, the recommended fix was implemented in **PR#147**.

[I-09] is-contract-deployer can be inlined

> **Severity:** *Informational* *(Resolved)*
>
> **Context:** ***governance.clar***:*168-173,605* ***meta-governance.clar***:*162-167,238*

Description

The function `is-contract-deployer` present in both the `governance` and `meta-governance` contracts verifies if the transaction sender is the initial contract deployed.

```
(define-private (is-contract-deployer)
  (begin
    (asserts! (is-eq tx-sender (var-get contract-deployer))
        ERR-NOT-CONTRACT-DEPLOYER)
    SUCCESS
  )
)
```

In both case this function is called only once, when `initialize-governance` is called and `initialize-governance` can only be called exactly one time per contract.

Recommendation

Because of the lack of reusability, the function can be removed and in the `initialize-governance` function calls, change the `(try! (is-contract-deployer))` call to a `(asserts! (is-eq tx-sender (var-get contract-deployer)) ERR-NOT-CONTRACT-DEPLOYER)` direct operation.

Resolution: Resolved, the recommended fix was implemented in **PR#155**.

[I-10] Removing a governance member does not invalidate his casted votes

> **Severity:** *Informational* *(Acknowledged)*
>
> **Context:** ***meta-governance.clar***:*110-116*

Description

When a member of governance is removed form the `meta-governance` contract, his already cast votes still remain for any ongoing proposal.

Recommendation

Document this behavior. Governance is considered trusted and even if a member becomes compromised and is kicked out, his proposals would eventually expire or never reach approval threshold.

Note, there is no proposal expiration mechanism in place, but that is further detailed in a different issue.

Resolution: Acknowledged by the team.

[I-11] There is no reserve_admin role

> **Severity:** *Informational* *(Resolved)*
>
> **Context:** *governance.clar*

Description

In regards to the protocol reserve, project documentation indicates that:

> The protocol reserve token balance can be accessed by a special role, called the `reserve_admin`, that initially coincides with the governance, but at a later point in time it can be assigned to a smart contract to enable extra features, like capital boosting, automatic liquidator payoff, or revenue distribution.

Currently, there is no such mechanism implemented in Granite.

Protocol reserve token balance management is done via creating a proposal in the `governance` contract and have governance members accept and execute it.

Execution is done via the governance actions `ACTION_DEPOSIT_TO_RESERVE` and `ACTION_WITHDRAW_FROM_RESERVE` but it requires the caller to be a member of governance. There is no mechanism to off-load the role to a special principal who is not a member of governance.

While the `reserve_admin` can be a contract and at the same time a member of governance it contradicts project documentation.

Recommendation

Modify the documentation to mention that currently the protocol reserve token balance can be accessed only by governance and that at a later point, support will be implemented into the `governance` contract as to allow a specific designated, non-governance, role to be able to manage the funds.

Resolution: Resolved, the recommended course of action was taken.

ABA: documentation has been updated to reflect current logic.

[I-12] state::toggle-upgrades function name is misleading

Severity: *Informational* *(Resolved)*

Context: ***state.clar***:*180-193*

Description

The `state` contract has a function `toggle-upgrades` which sets the status of upgradability to true or false, depending on the provided `status` flag by the governance when it is called.

The `toggle` word implies switching from one state to another. Meaning that one call should result in the change of the state, however the `toggle-upgrades` function is a setter function, setting the underlying value to a desired status. It is not a toggle, but a set.

Recommendation

Change the name of the `toggle-upgrades` function to a more correct and suggestive name, such as `set-upgrades-enabled`. Or modify it to work as a toggle, and flip the underlying feature state.

Resolution: Resolved, the recommended fix was implemented in **PR#143**.

[I-13] Remove debug remnants

Severity: *Informational* *(Resolved)*

Context: ***borrower.clar***:*32,237* ***governance.clar***:*273* ***liquidator.clar***:*55,235*

Description

Throughout the codebase there are minor debut remnants leftover.

In Clarity calling an arbitrary contract is done only if you pass a principal that implements a trait as input. Granite does not support this, as each market token asset needs to be hardcoded. For the purpose of testing and development, the current mocks were used:

- the market token contract is named `mock-usdc` in every contract that there is.
- the oracle contract that is used is `mock-oracle`. When the `pyth-oracle` is finished, after Nakamoto, this also needs to be changed

Recommendation

Acknowledge the issue and implement the mentioned changes before going live.

Resolution: Resolved, the recommended course of action was taken.

[I-14] Fully liquidating a collateral position does not remove the user collateral principal

Severity: *Informational* *(Acknowledged)*

Context: ***state.clar****:771-813*

Description

When a user is liquidated, an amount of collateral tokens are taken by the liquidator. If the full amount of tokens that users has deposited for a specific collateral is taken, the collateral principal is however left in the `user-collaterals` map with a position of 0 tokens and also in the `positions` map as a token users has collateral for.

A `user-collaterals` position with 0 amount does not influence the system asides from extra fees on execution. Also, users can then call `remove-collateral` with a 0 amount as to effectively remove the token.

Recommendation

Document the exact behavior when a liquidation is done in regard to 0 remaining collateral tokens.

If debt socialization is to be implemented, as mentioned in another issue, this finding may be invalidated depending on the chosen implementation.

Resolution: Acknowledged by the team.

[I-15] Apply the extended Clarity Style Guide

Severity: *Informational* *(Resolved)*

Context: Global

Description

There are styling optimizations that can be done as to generally reduce source code size while maintaining readability. These have been described in the **Extended Clarity coding style** detailed in this document.

Recommendation

Apply the recommendations in the guide.

Resolution: Resolved, the recommended fix was implemented in **PR#files**.

[I-16] linear-kinked-ir::accrue-interest can be simplified

Severity: *Informational* *(Resolved)*

Context: ***linear-kinked-ir.clar****:102-139*

Description

The `linear-kinked-ir::accrue-interest` function has several redundancies that can be removed.

- change all `tuple` declarations to using the curly braces version
- `elapsed-blocks` is declared but never used. Eider use it or remove it
- `new-lp-interest`, `new-protocol-interest` and `new-total-assets` are declared but only used as return values, once. Inline them.
- convert all spaces to tabs

Recommendation

Implement the mentioned suggestions.

Resolution: Resolved, the recommended fix was implemented in **PR#155**.

[I-17] state::increase-total-assets must never be executed permissionless

> **Severity:** ***Informational*** *(Resolved)*
>
> **Context:** ***state.clar****:664-670*

Description

The `state::increase-total-assets` permits allowed contracts to directly increase the total assets accounted for in the contract without minting shares through the `liquidity-provider` flow.

This function must never be linked to a contract where users can permissionless call it, as it would open up the vaults to an inflation attack.

Recommendation

Document the `state::increase-total-assets` security concerns, specifically mentioning to never have it indirectly permissionless called.

Resolution: Resolved. Fix was implemented in **PR#178**.

ABA: the function was modified to be callable only on testnet.

[I-18] Typographical errors

> **Severity:** ***Informational*** *(Resolved)*
>
> **Context:** ***pyth-oracle.clar****:29* ***governance.clar****:321,370,427,470,479,488,503,514,523,532,549,558,577,607* ***liquidator.clar****:49,135*

Description

Throughout the codebase there are instances of typographical errors.

- `pyth-oracle`: change `"maintainence"` to `"maintenance"`
- `governance`:

  - "percetage" to "percentage"
  - "guardinas" to "guardians"
  - "try to execure the propsal if threshold is met" to "try to execute the proposal if threshold is met"

- liquidator:
  - "premuim" to "premium"
  - "liqudator" to "liquidator"

Recommendation

Fixe the typos or remove the comments where they do not help.

Resolution: Resolved, the recommended fix was implemented in **PR#155**.

[I-19] English dialect inconsistencies

> **Severity:** *Informational* *(Resolved)*
>
> **Context:** Global

Description

A common best practice is to use one language and dialect for the sake of consistency, readability and maintainability. Throughout the codebase there are instances where both British and American dialect is used.

Example:

- the British *"utilisation"* with *"s"* is used in some places (e.g. `linear-kinked-ir::ir-calc`) while the American *"utilize"* version with *"z"* is used in other places (e.g. `linear-kinked-ir::utilization-calc`)
- the British *"initialise"* is used in some places (e.g. in `meta-governance`) while the American *"initialize"* version, with a *"z"* is used in other places (e.g. the `ERR-CONTRACT-ALREADY-INITIALIZED` error)

Recommendation

Consider keeping consistency and only using American or British English dialect.

Resolution: Resolved, the recommended fix was implemented in **PR#158**.

[I-20] Add descriptive error code for when no debt is present but health check is done

> **Severity:** *Informational* *(Resolved)*
>
> **Context:** ***liquidator.clar****:160*

Description

Calling the `liquidator::account-health` function when a user has no debt results in a runtime `DivisionByZero` error which cannot be caught by external integrators.

The error appears when the `position-health` variable is calculated.

Recommendation

Consider creating a new error constant for when user has no debt and validating that if `current-debt` is 0 then revert with that error message.

Returning an arbitrary value to signal that account is healthy (e.g. 1) is another approach.

Resolution: Resolved, the recommended fix was implemented in **PR#159**.

[I-21] liquidator::liquidate-collateral can be simplified in conjunction with liquidator::account-health

> **Severity:** ***Informational*** *(Resolved)*
>
> **Context:** ***liquidator.clar****:28,38-43,158-161*

Description

The `liquidator::liquidate-collateral` has several redundant or duplicated operations that can be simplified or removed:

- the `account-health` function already calculates the current-debt-adjusted variable which can be passed all the way to the `liquidate-collateral` function in the `position-data` variable. By doing this, the following code can be removed

```
(position (get user-position position-data))
;; get user current debt
(current-debt (contract-call? .math convert-to-debt-assets (get debt-shares
    position) true))
(current-debt-adjusted (contract-call? .math get-market-asset-value current-debt))
;; total collaterals value * collateral liquid ltv
(position-collaterals (get collaterals position))
```

- `(position-collaterals (get collaterals position))` is never used, it also can be removed

Recommendation

Implement the mentioned changes

Resolution: Resolved, the recommended fix was implemented in **PR#172**.

[I-22] Liquidation discount factor may be counterproductive in some conditions

Severity: ***Informational*** *(Resolved)*

Context: Global

Description

Granite implements a discount premium mechanism so that liquidators are incentivized to liquidate an underwater position and receive collateral worth more than the repaid amount in liquidations by the mentioned discount premium.

The OpenZeppelin's 2019 Compound audit, "Counterproductive Incentives" section, shows that any liquidation system that incentivizes liquidators with discounted collateral can, in some circumstances, leave violators more unhealthy than they were pre-liquidation. OpenZeppelin's 2020 Aave Protocol Audit also mentions the same issue being present.

Recommendation

The issue has existed in projects for several years as there is no straightforward solution to it. Reducing or removing the liquidation incentive would bring protocol insolvency due to lack of liquidation incentives. The issue also is not that impactful or often reached by users in a normal happy path scenario.

The newly-launched Euler V2 mitigates the issue with a combination of a dynamic discount factor with a maximum value. Even in this case, however, the mitigation depends on setting the right configurable limitations. More on this can be read in the Euler Finance Dutch Liquidation Analysis report.

For Granite V1, for now, this type of solution would be over-engineering, bringing more complexity to the code then the potential gains.

Acknowledge this issue and monitor for such situations. If the market shows an actual impact over time, in the next iteration of the protocol implement a mechanism similar to Euler's solution.

Resolution: Resolved, the recommended course of action was taken.

[I-23] linear-kinked-ir::utilization-calc can be simplified

Severity: ***Informational*** *(Resolved)*

Context: ***linear-kinked-ir.clar****:141-160*

Description

The `utilization-calc` function, which is used to determine the current utilization rate based on the existing total assets and open interest, has several redundancies and can be simplified to reduce code complexity and execution fees.

All the comments can be removed and one single comment be left before the `define-read-only` that indicates that *total-assets and open-interest are fixed u8*

The initial increasing of precision from 8 to 12 on the `open-interest` and `total-assets` inputs is redundant as it will be canceled out in the `(/ (* open-interest-fixed one-12) total-assets-fixed)` calculation. The ending precision/decimals of the result is define by the multiplication in the last calculus, meaning by the `one-12`.

This can be easily proved by changing the `open-interest-fixed` and `total-assets-fixed` precision to any arbitrary value between 8 (minimum) and 23 (maximum, as to not overflow with the current test values) and seeing that all tests pass.

Since both inputs are `uint`, the minimum they can be is 0, thus `(is-eq <condition> u0) u0 <operation>)` can also be shortened to `(> <condition> u0) <operation> u0)`.

Recommendation

Add the mentioned comment and remove the `to-fixed` intermediary variables and rewrite the function as suggested. For example, the entire `utilization-calc` function ca be rewritten as:

```
(define-read-only (utilization-calc (total-assets uint) (open-interest uint)) ;;
    fixed u8
  (if (> (+ total-assets open-interest) u0) (/ (* open-interest one-12)
      total-assets) u0)
)
```

Resolution: Resolved, the recommended fix was implemented in **PR#171**.

[I-24] Remove redundant comments

Severity: *Informational* *(Acknowledged)*

Context: ***bundler.clar***:*61-62,61-62* ***linear-kinked-ir.clar***:*28,56,84,91,149-150,152,170-171,214,236,238-239* ***pyth-oracle.clar***:*21-22,29* ***borrower.clar***:*172* ***governance.clar***:*4-65,72,94-162,470-558* ***liquidator.clar***:*34-56* ***meta-governance.clar***:*5-26,52-55,63,191*

Description

Throughout the codebase there are redundant comments. As Clarity smart contract source code size impacts execution performance, there should be no redundant comments within the codebase.

Recommendation

Remove the mentioned comments.

Resolution: Acknowledged by the team.

We prefer to keep comments for code clarity, also considering that soon Stacks will have WASM compatibility.

[I-25] Inconsistent contract deployer handling throughout the codebase

Severity: *Informational* *(Resolved)*

Context: ***linear-kinked-ir.clar****:35* ***governance.clar****:157* ***meta-governance.clar****:35*

Description

In Granite, some contracts require knowing who de deployer was. These are:

- `linear-kinked-ir:` `(define-constant contract-launcher contract-caller)`
- `meta-governance:` `(define-data-var contract-deployer principal tx-sender)`
- `governance:` `(define-data-var contract-deployer principal tx-sender)`

There is both a lack of uniformity in the naming, as some are `contract-deployer` while another is `contract-launcher` and a redundancy because in the `governance` and `meta-governance` contracts, the deployers are defined as data variables when in fact, they never changed.

Another slight aesthetic difference is that the `contract-launcher` is defined to the `contract-caller` while in the other two cases the values are to `tx-sender`. Functionally this does not matter as they both point to the same standard principal (address) when a contract is deployed.

Recommendation

In all three places change the declarations, so they are all called the same, are all constants and all are equal to `contract-caller`. Example: `(define-constant contract-deployer contract-caller)`

Resolution: Resolved, the recommended fix was implemented in **PR#161** and **PR#177**.

[I-26] Redundant safe-div call in liquidator::calc-collateral-to-give

Severity: *Informational* *(Resolved)*

Context: ***liquidator.clar****:193*

Description

In the `liquidator` contract, the `safe-div` function is used to send an error code, instead of a panic error, when a division by 0 is done.

In the `calc-collateral-to-give` function, however, it is redundantly used in the first division for calculating `repay-amount-with-discount`, as the `resolution-factor` cannot be 0, it is a constant declared in `state` and passed as such up to this point.

Recommendation

Use a normal division operator (/) when calculating the `repay-amount-with-discount` amount.

Resolution: Resolved, the recommended fix was implemented in **PR#160**.

[I-27] Check fees and limitations after pyth oracle deployment

> **Severity:** ***Informational*** *(Resolved)*
>
> **Context:** Global

Description

On Stacks, like the majority of blockchains, every transaction executed on-chain comes at a cost. A detailed description of these costs and limitations can be found in the Clarity of Mind: Runtime cost analysis chapter.

Granite will use the pyth oracle once the official is deployed on Stacks mainnet, possibly after the Nakamoto upgrade. Pyth uses a mechanism in which the callers submit valid price updates in a binary format call Verified Action Approvals (VAA).

These can become arbitrary large, the more prices are to be updated. Currently, Granite allows users to have up to 10 collaterals. Taking that into consideration, plus the market token, at the very limit of functioning the pyth VAA binary update blob will contain data for 11 tokens.

The current implementation allocates 8192 bytes for the VAA at maximum `(pyth-price-feed-data (buff 8192))`. This is enough, as tests have shown that for 11 collaterals, only 4351 bytes would be needed. Test could be done as the input VAA is retrieved by their Hermes system and interaction is already possible.

However, the Stacks execution fee costs and limitations could not be checked at this time.

Recommendation

Acknowledge this finding and after the official pyth version is deployed on Stacks mainnet, validated that the execution of an 11 token price update operation alongside liquidations, borrowing and removing collateral is possible.

Resolution: Resolved, the recommended course of action was taken.

The team will conduct more advanced testing once Pyth is available on Stacks Nakamoto testnet.

[I-28] Consider applying an anchoring pattern to the scaling-factor constant

> **Severity:** ***Informational*** *(Resolved)*
>
> **Context:** Global

Description

Throughout the codebase the `state` contract, `scaling-factor` constant is continuously passed as needed. This adds significant overhead both in complexity and call fees.

Recommendation

Create a separate contract named `constants` with only constants and getters for them. In this case, only the `scaling-factor`. In each contract where `scaling-factor` was used, now, define a local constant as:

```
(define-constant scaling-factor (contract-call?  .constants get-
scaling-factor))
```

Doing it as so, we "anchor" the constant to its original value in every contract without doing more than 1 call while also simplifying the code.

Resolution: Resolved, the recommended fix was implemented in **PR#files** and **PR#177**.

4 Applicability of the Zest Protocol Hack on Granite

On 11th April 2024, Zest Protocol, a prominent borrowing and lending protocol on Stacks, experienced a hack resulting in the loss of 322k STX tokens. The attack has since been mitigated and user balances have been restored by the protocol team from their treasury. Kudos to the team for their swift action.

During the current security audit of the Granite protocol, a thorough review was conducted of all security-related aspects. One specific question that arose was whether Granite could be susceptible to a similar attack as the one on Zest. While no security audit can guarantee the absence of bugs, only their presence, specifically for the attack on Zest, it could be determined whether Granite could have been affected by the same vulnerability.

The audit concluded that Granite would not have been affected by the vulnerability as it uses a fundamentally different approach to how collaterals are passed and stored in the system. Even without the current audit, the pre-audit version of the codebase could not of been hacked with the same type of attack as Zest.

To provide some context, we will first briefly detail the Zest hack.

On Zest, the attacker exploited a lack of collateral validation to borrow more assets than their actual collateral permitted. The way Zest was designed required the borrower to pass a list of up to 100 different collateral tokens, and for each provide a corresponding amount. Due to inadequate validation for duplicated tokens, an attacker could submit the same collateral token principal (address) 100 times but the amount provided only once, while the system would interpret it as if the collateral amount was correctly provided 100 times.

Coming back to Granite, this type of attack is mitigated because Granite does not require users to pass a list of collateral, it stores the collateral principal internally. Also, users add collateral by calling the `borrower:add-collateral` function which checks if the collateral principal is already in the list of user collaterals. Only if it is not present it is added.

```
;; check if the collateral is already in the list, otherwise add it
(updated-collaterals (if (is-none (index-of (get collaterals position)
    collateral-token))
  (unwrap! (add-item (get collaterals position) collateral-token) ERR-LIST-OVERFLOW)
  (get collaterals position)
))
```

To conclude, due to its protocol architectural structure and input validation processes, Granite is not vulnerable to the same type of attack that leverages inadequate collateral duplication checks.

More information on the Zest Protocol hack itself can be found in the protocol's security update, their public announcement and an in-depth analysis of the attack can also be found here.

5 Extended Clarity coding style

The following suggestions are oriented at reducing the source code size of a Clarity smart contract while maintaining readability or with a slight, but acceptable, impact on readability.

This is relevant in the context of Stacks smart contracts as, up until Clarity v2, whenever an external call is done, the entire source code of the called contract is copied into memory. On Stacks, there are also clear limitations regarding memory reads and writes per block.

It is also intended as an extension to the already existing Clarity of Mind: Coding style and Clarity of Mind: Runtime cost analysis book chapters.

Note, after the Nakamoto release and the introduction of `ClarityWASM` some suggestions may be outdated. Example abundant comments should be written at that point.

[S-01] Use tabs instead of spaces

Using spaces instead of tabs has the effect of increasing code size. Tabs visually can be interpreted as any number of spaces the viewer, generally 2 or 4. Setting aside how tabs are interpreted, tabs occupy only 1 byte of storage.

In contracts, an equivalent 2-spaces indent, while visually it can be the same, it occupies 1 byte per space character, thus double the storage space for a 2 space indent versus a tab.

[S-02] Set newline to LF instead of CRLF if using Windows OS

Windows OS uses CRLF (`\r\n`) to denote the newline character, while Unix-like systems represent it using LF (`\n`). Depending on the developer local git configuration, the CRLF of a Windows user that modifies the code will be saved in the repo.

Ensure all Clarity projects use the Unix style line endings to reduce code size cost by 50% (1 byte instead of 2) for every newline.

Note: CR: stands for Carriage Return (0x0D), LF stands for Line Feed (0x0A)

[S-03] Keep the function name on the same level as the define-*

Example:

```
(define-public
  (calculate (data (buff 8192)))
  ;; ... code ...
)
```

becomes:

```
(define-public (calculate (data (buff 8192)))
  ;; ... code ...
)
```

This reduces code size by the amount of indent plus newline character, while maintaining code readability.

[S-04] Keep all code as one line if it is not over 120 - 140 characters

In the early days of programming, as monitor screens were smaller than the ones that exist today, the common practice was user at maximum 70 - 80 characters.

With today's technology and editors, that limit is no longer valid.

Example:

```
(define-public (calculate
    (data (buff 8192))
    (token <token>)
    (amount uint)
  )
  (begin
    ;; ... code ...
  )
)
```

becomes:

```
(define-public (calculate (data (buff 8192)) (token <token>) (amount uint))
  (begin
    ;; ... code ...
  )
)
```

It reduces code size by the amount of indent plus newline characters, while maintaining code readability. Either 120 or 140 can be used as upper limits, depending on team preference.

[S-05] Separate functions or code using only one newline

Within the codebase, there should not be more then one newline between functions or, if within a function, one newline to separate regions for better code readability.

[S-06] Code region delimitations should be one line

When separating functions, code or logic within a `.clar` file, use only one comment line as to not redundantly increase size.

Example:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; READ-ONLY FUNCTIONS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

becomes:

```
;;;;;;;;;;;;; READ-ONLY FUNCTIONS ;;;;;;;;;;;;;
or
;; READ-ONLY FUNCTIONS
```

[S-07] Always use the use curly braces version of tuple declaration

In Clarity, tuples can be declared either using the `tuple` keyword or a dictionary-like definition using curly braces `{ ... }`

Avoid using the `tuple` declaration as it increases code size with little to no benefit, as the curly braces type declaration is both smaller and easier to understand for developers due to its similarity with other data constructs in other programming languages.

Example:

`(tuple (name "blockstack"))` becomes `{name: "blockstack"}`

or

```
(try! (contract-call? .state update-state (tuple
      (user tx-sender)
      (debt total-debt)
      (collaterals collateral-tokens)
      (shares shares)
      (amount amount)
)))
```

becomes:

```
(try! (contract-call? .state update-state {
      user: tx-sender,
      debt: total-debt,
      collaterals: collateral-tokens,
      shares: shares,
      amount: amount
}))
```

[S-08] Debug print commands can be compressed to one line even if surpassing 120 characters per line

The print function is used for development purposes or debug printing.

Functionally it generally has no impact with the exception when some off-chain systems use the information. Purely debug prints can be combined into one line, even if that makes them completely unreadable, as no meaningful logic is within them.

Example:

```
(print {
    collateral: collateral-token,
    liquidator: tx-sender,
    user: user,
    liquidated-collateral-amount: collateral-to-give,
    repaid-amount: repay-amount,
    repaid-shares: paid-shares,
    action: "liquidate-collateral"
})
```

becomes:

```
(print {collateral: collateral-token, liquidator: tx-sender, user: user,
   liquidated-collateral-amount: collateral-to-give, repaid-amount: repay-amount,
   repaid-shares: paid-shares, action: "liquidate-collateral"})
```

[S-09] Group ending parentheses and curly braces only at the end of a higher level command and indent to the higher command

Within the context of multiple commands, have the ending `)})  ...` be on the same level with the closes logical end, without losing meaning, when applicable.

Examples to better illustrate:

```
(try! (contract-call? .state update-borrow-state {
      user: tx-sender,
      user-debt-shares: total-user-debt-shares,
      user-collaterals: user-posted-collaterals,
      shares: shares,
      amount: amount
    }
  )
)
```

becomes:

```
(try! (contract-call? .state update-borrow-state {
      user: tx-sender,
      user-debt-shares: total-user-debt-shares,
      user-collaterals: user-posted-collaterals,
      shares: shares,
      amount: amount
}))
```

Note, do not put the `}))` as in the case above after the `amount`, because it makes readability harder.

More examples:

```
(define-private (accrue-interest)
  (let (
      (accrue-interest-params (unwrap! (contract-call? .state
          get-accrue-interest-params) ERR-INTEREST-PARAMS))
      (accrued-interest (try! (contract-call? .linear-kinked-ir accrue-interest
        (get last-accrued-block accrue-interest-params)
        (get lp-interest accrue-interest-params)
        (get protocol-interest accrue-interest-params)
        (get protocol-reserve-percentage accrue-interest-params)
        (get total-assets accrue-interest-params)))
      )
    )
    (contract-call? .state set-accrued-interest accrued-interest)
  )
)
```

becomes:

```
(define-private (accrue-interest)
  (let (
      (accrue-interest-params (unwrap! (contract-call? .state
          get-accrue-interest-params) ERR-INTEREST-PARAMS))
      (accrued-interest (try! (contract-call? .linear-kinked-ir accrue-interest
        (get last-accrued-block accrue-interest-params)
        (get lp-interest accrue-interest-params)
        (get protocol-interest accrue-interest-params)
        (get protocol-reserve-percentage accrue-interest-params)
        (get total-assets accrue-interest-params)
      )))
    )
    (contract-call? .state set-accrued-interest accrued-interest)
))
```

To note that the 3 `)))` were not moved at the end of the `(get total-assets accrue-interest-params)` but on the same indent as first beginning high level command, `(accrued-interest`

The `)` for the let is not touched as combining it with the other 3 `)` would result in decreasing code readability because there are commands after the end of the let. If the let was the last, it could have been combined.

Example (just applying the `)}` grouping, not all the indicated suggestions up to now):

```
(define-private (remove-user-collateral (prev-amount uint) (amount uint)
    (collateral principal) (debt-shares uint) (position-collaterals (list 10
    principal)))
  (let ((remaining-amount (unwrap! (contract-call? .math sub prev-amount amount)
      ERR-INSUFFICIENT-BALANCE)))
    (if (is-eq remaining-amount u0)
      (let ((updated-position-collaterals (remove-item position-collaterals
          collateral)))
        ;; remove the collateral since there is no user collateral left
        (try! (contract-call? .state remove-user-collateral tx-sender collateral
            debt-shares (get new-list updated-position-collaterals)))
        (ok (tuple
            (remaining-amount remaining-amount)
            (position-collaterals (get new-list updated-position-collaterals))
          )
        )
      )
      (begin
        ;; decrease the amount of collateral deposited by the user
        (try! (contract-call? .state update-user-collateral tx-sender collateral
            remaining-amount))
        (ok (tuple
            (remaining-amount remaining-amount)
            (position-collaterals position-collaterals)
          )
        )
      )
    )
  )
)
```

becomes:

```
(define-private (remove-user-collateral (prev-amount uint) (amount uint)
    (collateral principal) (debt-shares uint) (position-collaterals (list 10
    principal)))
  (let ((remaining-amount (unwrap! (contract-call? .math sub prev-amount amount)
      ERR-INSUFFICIENT-BALANCE)))
    (if (is-eq remaining-amount u0)
      (let ((updated-position-collaterals (remove-item position-collaterals
          collateral)))
        ;; remove the collateral since there is no user collateral left
        (try! (contract-call? .state remove-user-collateral tx-sender collateral
            debt-shares (get new-list updated-position-collaterals)))
        (ok (tuple
            (remaining-amount remaining-amount)
            (position-collaterals (get new-list updated-position-collaterals))
        ))
      )
      (begin
        ;; decrease the amount of collateral deposited by the user
        (try! (contract-call? .state update-user-collateral tx-sender collateral
            remaining-amount))
        (ok (tuple
            (remaining-amount remaining-amount)
            (position-collaterals position-collaterals)
        ))
      )
)))
```

Adding the `begin` end) to the ending e))) would have reduced readability.

Again, warning as to not abuse this suggestion, because if used incorrectly it will greatly decrease code readability.

Examples of applying the above suggestions to the Granite codebase

If we apply all suggestions (plus creating a variable for the duplicated `get new-list updated-position-collaterals` operation) on the above `remove-user-collateral` function, we get the following code:

```
(define-private (remove-user-collateral
    (prev-amount uint)
    (amount uint)
    (collateral principal)
    (debt-shares uint)
    (position-collaterals (list 10 principal))
  )
  (let ((remaining-amount (unwrap! (contract-call? .math sub prev-amount amount)
      ERR-INSUFFICIENT-BALANCE)))
    (if (is-eq remaining-amount u0)
      (let (
          (updated-position-collaterals (remove-item position-collaterals
              collateral))
          (updated-collaterals (get new-list updated-position-collaterals))
        )
        ;; remove the collateral since there is no user collateral left
        (try! (contract-call? .state remove-user-collateral tx-sender collateral
            debt-shares updated-collaterals))
        (ok {remaining-amount: remaining-amount, position-collaterals:
            updated-collaterals})
      )
      (begin
        ;; decrease the amount of collateral deposited by the user
        (try! (contract-call? .state update-user-collateral tx-sender collateral
            remaining-amount))
        (ok {remaining-amount: remaining-amount, position-collaterals:
            position-collaterals})
      )
)))
```

Another example for the `get-ir` function:

```
(define-read-only
  (get-ir
      (total-assets uint)
      (open-interest uint)
  )
  (begin
    (asserts! (var-get is-initialized) ERR-NOT-INITIALIZED)
    (ok
      (ir-calc ;; interest
        (utilization-calc total-assets open-interest) ;; utilization
      )
    )
  )
)
```

becomes:

```
(define-read-only (get-ir (total-assets uint) (open-interest uint))
  (begin
    (asserts! (var-get is-initialized) ERR-NOT-INITIALIZED)
    (ok (ir-calc (utilization-calc total-assets open-interest)))
  )
)
```

6 Disclaimer

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts the consultant to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. The consultant’s position is that each company and individual are responsible for their own due diligence and continuous security. The consultant’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology that is analyzed.

The assessment services provided by the consultant is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. Furthermore, because a single assessment can never be considered comprehensive, multiple independent assessments paired with a bug bounty program are always recommend.

For each finding, the consultant provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved, but they may not be tested or functional code. These recommendations are not exhaustive, and the clients are encouraged to consider them as a starting point for further discussion.

The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties. Notice that smart contracts deployed on the blockchain are not resistant from internal/external exploit. Notice that active smart contract owner privileges constitute an elevated impact to any smart contract’s safety and security. Therefore, the consultant does not guarantee the explicit security of the audited smart contract, regardless of the verdict.

The consultant retains the right to re-use any and all knowledge and expertise gained during the audit process, including, but not limited to, vulnerabilities, bugs, or new attack vectors. The consultant is therefore allowed and expected to use this knowledge in subsequent audits and to inform any third party, who may or may not be our past or current clients, whose projects have similar vulnerabilities. The consultant is furthermore allowed to claim bug bounties from third-parties while doing so.