ClarityAlliance

ZEST PROTOCOL v2 (Upgrade V2) SECURITY REVIEW

Conducted by:
KRISTIAN APOSTOLOV, ALIN BARBATEI (ABA), SILVEROLOGIST

DECEMBER 20TH, 2025

Contents

1. About Clarity Alliance

Clarity Alliance is a team of expert whitehat hackers specialising in securing protocols on Stacks.

They have disclosed vulnerabilities that have saved millions in live TVL and conducted thorough reviews for some of the largest projects across the Stacks ecosystem.

Learn more about Clarity Alliance at clarityalliance.org.

2. Disclaimer

This report is not, and should not be considered, an endorsement or disapproval of any project, team, product, or asset, nor does it reflect on their economics, value, business model, or legal compliance. It does not provide any warranty regarding the absolute bug-free nature or functionality of the analyzed technology.

Nothing in this report should be used to make investment or participation decisions. It does not constitute investment advice. Instead, it reflects an extensive assessment process intended to help clients improve code quality and reduce the inherent risks associated with cryptographic tokens and blockchain systems.

Blockchain technology presents ongoing and significant risk, and each company or individual remains responsible for their own due diligence and security posture. Clarity Alliance aims to reduce attack vectors and technological uncertainty but does not guarantee the security or performance of any system we review.

All assessment services are subject to dependencies and active development. Your access to and use of any services, reports, or materials is at your sole risk on an as-is and as-available basis.

Cryptographic tokens are emergent technologies with high technical uncertainty, and assessment results may include false positives, false negatives, or other unpredictable outcomes. Smart contracts may depend on multiple external parties, remain vulnerable to internal or external exploitation, and may carry elevated risks if owner privileges remain active. Accordingly, Clarity Alliance does not guarantee the explicit security of any audited smart contract, regardless of the reported verdict.

3. Introduction

A time-boxed security review of Zest protocol, where Clarity Alliance reviewed the scope and provided insights on improving the protocol.

4. About Zest Protocol

Zest Protocol is the DeFi protocol built for Bitcoin. Fully on-chain and open-source, it is building the future of Bitcoin inance.

We've launched Zest Protocol Borrow, enabling users to unlock liquidity by borrowing against their assets.

Live on Stacks - the leading Bitcoin Layer - Zest is now the top DeFi protocol on the network. Through the Stacks Market, users can deposit idle assets such as STX, sBTC, stSTX, USDC, and others to earn yield, accumulate points, and access overcollateralized loans..

Zest exists to make Bitcoin productive - every sat of it. The goal is to build a vibrant borrowing and lending ecosystem around Bitcoin as an asset.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

Scope

The following contracts, located in the **zest-core** repository, were in scope of the security review:

- `dao\dao-multisig.clar`
- `dao\dao-executor.clar`
- `dao\dao-treasury.clar`
- `dao\traits.clar`
- `market\market.clar`
- `market\market-vault.clar`
- `registry\egroup.clar`
- `registry\assets.clar`
- `registry\reserve-calculator.clar`
- `vault\vault-stx.clar`
- `vault\vault-sbtc.clar`
- `vault\vault-ststx.clar`
- `vault\vault-usdc.clar`
- `vault\vault-usdh.clar`
- `vault\traits.clar`

Initial Commit Reviewed

`fab7cdf569b4165b2c0bd47fd7ff46717d5e8b43`

Final Commit After Remediations

`3e5c24c187d5212e72a4a89ab960cb2f5a5a2608`

7. Executive Summary

Over the course of the security review, Kristian Apostolov, Alin Barbatei (ABA), Silverologist engaged with - to review Zest Protocol. In this period of time a total of **15** issues were uncovered.

Protocol Summary

Protocol Name	Zest Protocol
Date	December 20th, 2025

Findings Count

Severity	**Amount**
Critical	1
Medium	7
Low	5
QA	2
Total Findings	**15**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Positions With ZTokens Cannot Be Liquidated If Not Already Cached	Critical	Resolved
[M-01]	DAO Treasury Cannot Transfer wSTX or stSTX	Medium	Resolved
[M-02]	wSTX Wrapper Issues	Medium	Resolved
[M-03]	Pyth Price Updates Cannot Be Bundled in a Single Stacks Transaction	Medium	Resolved
[M-04]	Egroups Non-Mask Configurations Can't Be Easily Changed	Medium	Resolved
[M-05]	Positions With No Debt May Be Forced To Leave Dust Collateral	Medium	Resolved
[M-06]	Users Can Enter a Market Position Without an Egroup	Medium	Resolved
[M-07]	Full Vault Asset Debt Socialization Bricks Vault Indefinitely	Medium	Acknowledged
[L-01]	Flash Loan Fee Disregards Deposit Cap	Low	Resolved
[L-02]	Vaults Use tx-sender for Authorization	Low	Resolved
[L-03]	Add Extra Protection Against Self-Liquidations	Low	Resolved
[L-04]	Use Intentional Unwrap Logic on Asset Retrieval Path	Low	Resolved

[L-05]	Previews Incorrectly Overprice Vault Shares	Low	Resolved
[QA-01]	Add Ability to Disable Borrowing Specific Assets Per EGroup	QA	Resolved
[QA-02]	Ambiguous redeem Panic Revert in Vault	QA	Resolved

8. Findings

8.1. Critical Findings

[C-01] Positions With ZTokens Cannot Be Liquidated If Not Already Cached

Location:

- market.clar#L488-L493
- market.clar#L1430

Description
Liquidations of positions that include zTokens fail if no caching has occurred within the same block.

This happens because the `market::liquidate` function first calls `get-liquidation-context`:

```
(context (try! (get-liquidation-context borrower)))
```

and only afterwards performs the caching:

```
;; accrue
(u-debt (accrue-user-debts (get debt pos-full)))
(u-coll (accrue-user-collateral (get collateral pos-full)))
```

Inside `get-liquidation-context`, the `get-assets` function is invoked:

```
(define-private (get-liquidation-context (account principal))
  (let ((position (try! (get-liquidation-position account))))
    (ok {
      position: position,
      assets: (get-assets (get mask position))
    })))
```

`get-assets` ultimately attempts to fetch asset prices. If any of these assets are zTokens, it calls `resolve-ztoken`, which fails with `ERR-ORACLE-CALLCODE` when no cache entry exists:

```
(define-private (resolve-ztoken (p uint) (aid uint))
  (let ((cached (unwrap! (get-cached-indexes aid) ERR-ORACLE-CALLCODE))
```

As a result, any user position that uses zTokens as collateral will cause the liquidation to revert if the zTokens have not already been cached by another market operation in the same block.

Recommendation

Split the `get-liquidation-context` function into two parts. The position lookup should occur before accrual (and before it is used), while the `get-assets` call should be moved to execute after accrual.

8.2. Medium Findings

[M-01] DAO Treasury Cannot Transfer wSTX or stSTX

Location:

- dao-treasury.clar#L33

Description

The current implementation of the `dao-treasury` contract does not allow tokens that rely on `tx-sender` for authorization (e.g., wrapped STX or `stSTX`) to be transferred out of it, because it does not use `as-contract` transfer logic:

```
(try! (contract-call? token transfer amount current-contract recipient none))
```

At present, all observed protocol operations transfer only LP tokens to the treasury as fees. However, if fees are ever sent in the underlying asset format, transfers of tokens that depend on `tx-sender` authorization will revert.

Recommendation

Implement an `as-contract?` transfer from the `dao-treasury`, checking for the underlying asset for wSTX and then applying the appropriate approval limits.

[M-02] wSTX Wrapper Issues

Location:

- wstx.clar

Description

The current wSTX wrapper contract contains a functional issue and several redundant operations that should be removed.

Issue: wSTX is not SIP-10 compliant

SIP-10 requires that the `transfer` function emit a `memo` if one is provided. However, the current implementation ignores the `memo` parameter entirely and does not emit it.

```
(define-public (transfer (amount uint) (sender principal)
  (recipient principal) (memo (optional (buff 34))))
  (begin
    (asserts! (or (is-eq tx-sender sender)
      (is-eq contract-caller sender)) err-not-token-owner)
    (stx-transfer? amount sender recipient)
  )
)
```

Redundant or improvable operations:

1. The comparison `(asserts! (or (is-eq tx-sender sender) (is-eq contract-caller sender)) err-not-token-owner)` is redundant, because the `stx-transfer?` call can only succeed if the sender is the `tx-sender`:

   sender must be current tx-sender.

This also implicitly returns the correct error value:

   (err u4) sender not tx-sender.

The `contract-caller` check is unnecessary, since even if it passes, `stx-transfer?` will still fail if `sender` is not exactly `tx-sender`.

The `err-not-token-owner` error can therefore be removed as well.

1. Leftover, unused code artifacts

The following code is unused and can be removed:

```
(define-constant err-unauthorised (err u3000))
(define-fungible-token wstx)

(define-public (mint (amount uint) (recipient principal))
  (err u1)
)

(define-public (burn (amount uint) (owner principal))
  (err u1)
)
```

Note that the currently used Zest v1 on-chain wSTX version has the same issues, but does not include the `burn` function.

1. All data-vars can be constants

Since none of the data variables are ever modified, they should be defined as constants to reduce integration complexity and costs for third parties:

```
(define-data-var token-name (string-ascii 32) "wSTX")
(define-data-var token-symbol (string-ascii 10) "wSTX")
(define-data-var token-uri (optional (string-utf8 256)) none)
(define-data-var token-decimals uint u6)
```

1. Use a more descriptive token name

The token name is currently `wSTX`, identical to the symbol. In most ecosystems, wrapper tokens follow the `"Wrapped <token-symbol>"` naming convention, which is clearer for users and integrators.

Recommendation

Update the `transfer` function to correctly handle and emit the `memo`.

Example implementations:

```
(define-public (transfer (amount uint) (sender principal)
  (recipient principal) (memo (optional (buff 34))))
  (begin
    (try! (stx-transfer? amount sender recipient))
    (match memo to-print (print to-print) 0x)
    (ok true)
  )
)
```

or

```
(define-public (transfer2 (amount uint) (sender principal)
  (recipient principal) (memo (optional (buff 34))))
  (match memo
    to-print (stx-transfer-memo? amount sender recipient to-print)
    (stx-transfer? amount sender recipient)
  )
)
```

Apply the remaining changes by removing unused code, converting data variables to constants, and renaming the token to `"Wrapped STX"`.

[M-03] Pyth Price Updates Cannot Be Bundled in a Single Stacks Transaction

Location:

- market.clar
- market-trait.clar

Description

With the latest changes, the `market` contract always treats the `contract-caller` as the primary participant account in all entry points. While this design is safer, it complicates integration for third-party tools that only intend to wrap logic.

The `contract-caller` security improvement was introduced specifically to prevent wrapping logic, which can be abused.

Previously, one such wrapping pattern was used to update Pyth price feeds, which users are expected to do. Under the current design, users or any third party must call the Pyth write feed updates in separate transactions.

This is undesirable given the project's stated business intent.

Recommendation

There are two possible approaches to resolve this issue:

1. Change the entry point logic from using `contract-caller` to `tx-sender`. This would be a security downgrade and would again allow wrapper contracts that update Pyth before calling the main entry points.

2. A more security-oriented solution is to add an optional list of feeds to each relevant entry point and, when provided, attempt to update Pyth with them.

Note: in the current system, only three feeds (STX, sBTC, and USDC) need to be updated via Pyth. However, this may change in the future, so the recommendation is to support a list with a capacity greater than three. Otherwise, third-party integrators would need to modify the `<market-trait>` at that point.

Example implementation:

```
(define-private (write-feed (feed (buff 8192)) (status (response bool uint)))
  (match status
    success-status
      (match
        (contract-call? 'SP1CGXWEAMG6P6FT04W66NVGJ7PQWMDAC19R7PJ0Y.pyth-oracle-v4 verify-and-u
          feed
          {
            pyth-storage-contract: .pyth-storage-v4,

                          pyth-decoder-contract: 'SP1CGXWEAMG6P6FT04W66NVGJ7PQWMDAC19R7PJ0Y.pyt

                          wormhole-core-contract: 'SP1CGXWEAMG6P6FT04W66NVGJ7PQWMDAC19R7PJ0Y.wo
          }
        )
        update-success (ok true)
        update-failed ERR-PRICE-FEED-UPDATE-FAILED)
    error-status status
  )
)

(define-private (write-feeds (feeds (optional (list 3 (buff 8192)))))
  (match feeds
    entries (fold write-feed entries (ok true))
    (ok true)))
```

and call it as `(feeds-check (try! (write-feeds price-feeds)))`.

[M-04] Egroups Non-Mask Configurations Can't Be Easily Changed

Location:

- egroup.clar#L449

Description

The current implementation allows changing egroup configuration via `egroup::update`.

However, this function rejects any update that does not modify the mask:

```
(asserts! (not (is-eq prev-MASK new-MASK)) (ok true))
```

As a result, if the team needs to change an LTV group, they must first change the mask to some different, arbitrary mask, and then change it back to the original mask along with the LTV modification for the update to be accepted.

Recommendation

Remove the `(asserts! (not (is-eq prev-MASK new-MASK)) (ok true))` check altogether.

[M-05] Positions With No Debt May Be Forced To Leave Dust Collateral

Location:

- market.clar#L1141-L1166

Description When a user removes collateral from the market via `collateral-remove`, their existing overall collateral value is rounded down:

```
;; LTV (enabled collaterals only)
(notional-valued-assets
  (get-notional-evaluation { position: position, assets: assets }))
(collateral-value (get collateral notional-valued-assets))
```

but the collateral to be removed is evaluated by rounding up:

```
(removed-asset-value
  (find-and-resolve-asset-value assets asset-id amount true)))
```

This is done to ensure no value leakage occurs against the protocol when removing assets.

However, an edge case arises where users with no debt are still subjected to a health check. For users attempting to withdraw their full collateral, this check may revert due to the rounding behavior described above.

This effectively forces users to leave dust as collateral in the market vault.

Recommendation

In the `market::collateral-remove` function, perform health checks only if the user has debt.

[M-06] Users Can Enter a Market Position Without an Egroup

Location:

- market.clar#L1068

Description
A key system invariant is that all users in the system must have their positions associated with an egroup.

The team is responsible for ensuring that every combination of actions that allows users to enter the system is backed by a corresponding egroup. In other words, if a user has added collateral, they may be allowed to borrow or remove collateral, but only if there is an egroup that supports that position.

Egroups should be broadly configured so that any permutation and variation of a user's position is covered by an egroup. In the unlikely case where, for example, the protocol supports 3 assets but there is no egroup that covers one of them, entry into the system with that unsupported configuration should be blocked.

The current implementation of `market::collateral-add` incorrectly skips the egroup check when the user is new, the collateral is new, and there is no associated debt.

This leads to the following problematic scenario:

- The team adds 2 assets as collateral.
- By mistake, they only add an egroup that supports 1 of those collateral assets.
- A user can add the first, supported asset, and then add the second, unsupported asset. Because the user has no debt, the `(future-group (try! (get-egroup future-mask)))` check is never reached:

```
;; ONLY check capacity if user has debt
(if (> current-debt-usd u0)
    ;; Calculate future mask and validate egroup exists
    (let ((current-coll-usd (get collateral current-notional))
          (current-capacity (* current-coll-usd current-ltv))
          (added-collateral-value (try! (get-asset-value asset amount false)))
          (future-group (try! (get-egroup future-mask)))
```

As a result, the position is allowed to be created.

From that point on, any protocol interaction will fail for the user because retrieving their egroup (an action performed on all entry points) will always revert with `ERR-NO-EGROUP-FOUND`. The user's funds will remain locked in the system until a new egroup is added that supports the user's position.

Recommendation

Move the `(future-group (try! (get-egroup future-mask)))` call from inside the `(if (> current-debt-usd u0)` branch to immediately after the `(future-mask (bit-or current-mask (pow u2 asset-id)))` declaration.

This will ensure that users cannot enter a position without an associated egroup.

[M-07] Full Vault Asset Debt Socialization Bricks Vault Indefinitely

Location:

- vault-sbtc.clar#L931

Description When socializing bad debt, vaults cap all value subtractions to the maximum value in the system, effectively capping each update at resetting the value back to zero.

```
;; socialize-debt
    (var-set principal-scaled (if (> scaled-principal scaled-amount)
      (- scaled-principal scaled-amount) u0))
    (var-set total-borrowed (if (> borrowed debt-reduction)
      (- borrowed debt-reduction) u0))
    (var-set assets (if (> current-assets debt-reduction)
      (- current-assets debt-reduction) u0))
```

This issue arises because `assets` can be set to zero when bad debt exceeds the protocol's reserves. Beyond completely wiping the share value of vault holders, this also bricks the vault's functionality due to a division-by-zero in `convert-to-shares-preview`:

```
(if (is-eq ts u0)
        amount
        (mul-div-down amount ts ta))))
```

Recommendation

Because this situation requires a catastrophic black swan event whose market impact would be far more detrimental to the protocol overall, the most practical approach is to acknowledge the bricking side effect on a fully insolvent vault and redeploy afterwards.

Additionally, `convert-to-shares-preview` and `convert-to-assets-preview` should be augmented with guard clauses on the denominator, so they return an error instead of causing a division-by-zero panic at runtime.

8.3. Low Findings

[L-01] Flash Loan Fee Disregards Deposit Cap

Location:

- vault-sbtc.clar#L991-L998

Description When a flash loan fee is paid by users, it is converted into LP tokens and transferred to the treasury.

```
;; Only mint treasury shares and add fee if fee > 0
(if (> fee u0)
    (let ((treasury-shares (convert-to-shares-preview fee)))
        (if (> treasury-shares u0)
            (try! (ft-mint? zft treasury-shares .dao-treasury))
            false)
        (var-set assets (+ (var-get assets) fee)))
    false)
```

This effectively behaves as a deposit, increasing the underlying assets while also minting shares.

Note that if the minting had not been performed, this would have been interpreted as a donation.

This behavior directly contrasts with the cap that is enforced on user deposits.

```
(define-public (deposit (amount uint) (min-out uint) (recipient principal))
    (let (
      ;; ... code ...
      (CAP-SUPPLY (var-get cap-supply))
      (current-assets (var-get assets))
    ;; ... code ...
    (asserts! (<= (+ current-assets amount) CAP-SUPPLY) ERR-SUPPLY-CAP-EXCEEDED)
```

Recommendation

Transfer the fee amount directly to the treasury without converting it to LP tokens and without accounting for it in the vault. Apply this change to all vaults.

[L-02] Vaults Use `tx-sender` for Authorization

Location:

- GLOBAL

Description
All vault implementations use `(account tx-sender)` in both the `deposit` and `redeem` functions.

This allows integrators to call `deposit` or `redeem` on behalf of the `tx-sender`. However, this behavior can be abused by malicious parties if a user is ever phished into signing a transaction that effectively empties all of their vaults.

Recommendation

Modify the `vault::deposit` and `vault::redeem` functions to use `contract-caller` as the account.
This will also require changes to the `vault::initialize` function, since all deposits must now be made in a context where `contract-caller` equals `tx-sender`; otherwise, the call will revert.

Currently, the `initialize` function is DAO-gated, and DAO gating requires `tx-sender` to be the `dao-executor`, while `contract-caller` will always be the proposal contract.

To address this, remove the DAO authorization check. There is no incentive for a third party to call `initialize`, since the initial deposit amount is locked to the null address regardless of the caller's intent.

[L-03] Add Extra Protection Against Self-Liquidations

Location:

- market.clar

Description
A well-known vulnerability in borrowing and lending protocols relates to self-liquidations triggered during market price updates.

An attack exploiting this would unfold as follows:

- The attacker monitors for an oracle price update transaction that will reduce the price of the collateral.
- The attacker then sandwiches the oracle price update transaction:
  - Front-runs it by adding collateral and borrowing against it.
  - Back-runs it by self-liquidating.
  - Finally, removes the remaining collateral.

Note that this attack is only viable if debt socialization exists.

The attack is profitable only if the attacker can recover the entire collateral plus an additional amount derived from the liquidation penalty. Flash loans are typically used to amplify profits. For systems relying on oracles like Pyth, as in the case of Zest, the attack is more feasible because anyone can update the price within a valid range of potential prices during a given time window (1 minute by default on the Pyth network).

This type of attack is also known as oracle frontrunning.

To mitigate this attack, Zest has already implemented differential loan-to-value (LTV) ratios. When a user borrows, their collateral is evaluated at a "borrow LTV," and when liquidated, the same collateral is evaluated at a higher LTV (partial liquidation LTV and full liquidation LTV). This design ensures that users cannot be instantly liquidated solely due to the valuation gap between the borrowing and liquidation valuations.

In addition to this existing mitigation, Zest can implement further mechanisms to reduce the effectiveness of such attacks.

Recommendation

Oracle frontrunning attacks cannot be fully mitigated. However, to reduce the risk, an additional mechanism can be introduced: disallowing a user from being liquidated in the same block in which they last borrowed.

This mechanism would specifically block flash-loan-based attacks, since an attacker would be unable to borrow, be liquidated, and repay the flash loan within the same transaction, even under an extreme price movement.

Because borrowing is only allowed when the collateral value, adjusted by the borrow (maximum) LTV, exceeds the debt valuation, the likelihood that a normal, honest user would avoid liquidation for one block in a legitimate scenario is extremely low and can be considered an acceptable trade-off.

[L-04] Use Intentional Unwrap Logic on Asset Retrieval Path

Location:

- assets.clar

Description Throughout the `market` contract, entry points first retrieve asset information via a call to `assets::get-asset-status`:

```
(define-private (get-asset (asset principal))
  (contract-call? .assets get-asset-status asset))
```

This function reverts with an unwrap panic if an invalid asset is passed, instead of returning a dedicated error code.

Recommendation

Update `assets::get-asset-status` and all related functions in the `assets` contract so they revert with a specific error code when an invalid asset is passed, rather than relying on an unwrap panic.

[L-05] Previews Incorrectly Overprice Vault Shares

Location:

- vault-sbtc.clar#L306-L311
- vault-sbtc.clar#L313-L318
- vault-sbtc.clar#L380-L381

Description

`convert-to-shares-preview` and `convert-to-assets-preview` are intended to preview share conversions based on interest that has not yet been accrued in the vault. They do this by using `total-assets-preview`, which accounts for unaccrued interest in the system.

```
(define-private (total-assets-preview)
  (let ((current-assets (var-get assets))
        (debt (debt-preview))
        (borrowed (var-get total-borrowed))
        (interest (if (> debt borrowed) (- debt borrowed) u0)))
    (+ current-assets interest)))
```

However, these functions do not account for the treasury LP share dilution fee mechanism when accruing, which causes them to consistently overprice shares by the amount owed to the treasury.

Since all mutating entry points of the vault accrue before calling the preview functions, this issue only affects readonly data.

Similarly, `principal-ratio-reduction` uses `total-debt` instead of `debt-preview`, which also results in incorrect readonly values.

Recommendation

Update the preview functions to take LP share dilution into account.

Example implementation for `convert-to-shares-preview` and `convert-to-assets-preview`:

```
(define-private (convert-to-assets-preview (amount uint))
  (let ((ta (total-assets-preview))
        (ts (total-supply-preview)))
    (if (is-eq ta u0)
        u0
        (mul-div-down amount ta ts))))

(define-private (convert-to-shares-preview (amount uint))
  (let ((ta (total-assets-preview))
        (ts (total-supply-preview)))
    (if (is-eq ts u0)
        amount
        (mul-div-down amount ts ta))))

(define-private (total-supply-preview)
  (let ((current-supply (total-supply))
        (treasury-lp (calc-treasury-lp-preview)))
    (+ current-supply treasury-lp)))

(define-private (calc-treasury-lp-preview)
  (let ((scaled-principal (var-get principal-scaled))
        (idx (var-get index))
        (next (next-index))
        (old-debt (mul-div-down scaled-principal idx INDEX-PRECISION))
        (new-debt (mul-div-down scaled-principal next INDEX-PRECISION))
        (debt-delta (if (> new-debt old-debt) (- new-debt old-debt) u0))
        (reserve-inc (mul-div-down debt-delta (var-get fee-reserve) BPS))
        (ta-preview (total-assets-preview)))
    (if (> reserve-inc u0)
        (mul-div-down reserve-inc (total-supply) (- ta-preview reserve-inc))
        u0)))
```

`principal-ratio-reduction`:

```
(define-private (principal-ratio-reduction (amount uint))
  (calc-principal-ratio-reduction amount (var-get principal-scaled)
    (debt-preview)))
```

8.4. QA Findings

[QA-01] Add Ability to Disable Borrowing Specific Assets Per EGroup

Location:

- egroup.clar

Description In practice, there will be situations where the protocol may wish to prevent users from borrowing a specific asset, but only while they are in a particular EGroup.

The current implementation does not support this functionality.

Recommendation

Add a flag to each EGroup registry entry to enable or disable borrowing. This flag should be checked in `market::borrow` to ensure that no further borrows are allowed when disabled.

From an implementation perspective, we suggest:

- In the LTV group registry, add a new `uint` field called `BORROW-DISABLED-MASK`.
- In this mask, each bit position corresponds to a debt asset ID, without any offset by 64.
- In each `market::borrow` call, retrieve this mask and check whether the bit for the specific asset is set. If it is set, revert the borrow:

```
(asserts! (is-eq (bit-and disabled-borrow-mask
  (pow u2 asset-id)) u0) ERR-LTV-GROUP-ASSET-BORROW-DISABLED)
```

Note: the implementation uses a `disable` mask instead of an `enable` mask because the most common and default case is for all borrowing to be enabled. Also, since each group may have different sets of borrowable assets, it is simpler not to require EGroup insertions to compute which assets to allow by default. Instead, passing `0u` indicates that no assets are blocked.

[QA-02] Ambiguous `redeem` Panic Revert in Vault

Location:

- vault-sbtc.clar#L787

Description
In the vault implementations, when a depositor wants to withdraw funds, they call the `redeem` function with the number of shares to burn.

The `redeem` function checks that the user has enough tokens to burn:

```
(asserts! (>= balance amount) ERR-INSUFFICIENT-BALANCE)
```

This is intended to revert with a more meaningful error code than the default burn error code of `1`:

`(err u1) -- sender does not have enough balance to burn this amount or the amount specified is not positive`.

However, the `convert-to-assets-preview` call, which is executed before this check, can revert with an arithmetic overflow if the value is sufficiently large (e.g., `MAX-UINT-128`).

In such cases, external integrators may have difficulty debugging the failed transaction.

Recommendation
Move the `(asserts! (>= balance amount) ERR-INSUFFICIENT-BALANCE)` check into an inlined `let` binding immediately after retrieving the balance. Apply this change to all vaults.